IN-60
43103
p 57

Sparse Distributed Memory Principles of Operation

M.J. Flynn
P. Kanerva
N. Bhadkamkar

December 1989

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 89.53

NASA Cooperative Agreement Number NCC 2-387

RIACS

Research Institute for Advanced Computer Science
An Institute of the Universities Space Research Association

Sparse Distributed Memory: Principles and Operation

M.J. Flynn, P. Kanerva,** and N. Bhadkamkar**

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 89.53
December 1989

Abstract. Sparse distributed memory is a generalized random-access memory (RAM) for long (e.g., 1,000 bit) binary words. Such words can be written into and read from the memory, and they can also be used to address the memory. The main attribute of the memory is sensitivity to similarity, meaning that a word can be read back not only by giving the original write address but also by giving one close to it as measured by the Hamming distance between addresses.

Large memories of this kind are expected to have wide use in speech recognition and scene analysis, in signal detection and verification, and in adaptive control of automated equipment---in general, in dealing with real-world information in real time.

The memory can be realized as a simple, massively parallel computer. Digital technology has reached a point where building large memories is becoming practical. This research project is aimed at resolving major design issues that have to be faced in building the memories. This report describes the design of a prototype memory with 256-bit addresses and from 8K to 128K locations for 256-bit words. A key aspect of the design is extensive use of dynamic RAM and other standard components.

*Work reported herein was conducted primarily at Stanford University, supported in part by the National Aeronautics and Space Administration (NASA) under contract NAGW 419 and was published concurrently at Stanford University as Technical Report CSL-TR-89-400.

**A portion of the work reported herein was conducted at RIACS, supported in part by Cooperative Agreement NCC 2-387 between NASA and the Universities Space Research Association (USRA).

Contents

List of Figures

List of Tables

Acknowledgements

We would like to thank the following people at Stanford and at RIACS who have contributed to the Sparse Memory project through design effort, advice, and writing portions of this report: Bahram Ahanin, Brian Flachs, Paul Flaherty, Philip Hickey, Alan Kobrin, André Lamothe, Eric Lochner, Kurt Webber, Bob Zeidman, and Andrew Zimmerman. Finally, Susan Gere assembled and edited the various versions of the document.

Chapter 1

Introduction to Sparse Distributed Memory

1.1 Introduction

Sparse distributed memory (SDM) is a generalized random-access memory (RAM) for long (e.g., 1,000 bit) binary words. These words serve as both addresses to and data for the memory. The main attribute of the memory is sensitivity to similarity, meaning that a word can be read back not only by giving the original write address but also by giving one close to it, as measured by the number of mismatched bits (i.e., the Hamming distance between addresses).

The theory of the memory is mathematically complete and has been verified by computer simulation. It arose from the observation that the distances between points of a high-dimensional space resemble the proximity relations between concepts in human memory. The theory is also practical in that memories based on it can be implemented with conventional RAM-memory elements. The memory array in the prototype memory makes extensive use of 1 M-bit DRAM technology, with array modules in concurrent execution. Consequently, the prototype is inexpensive compared to implementations of the memory on systolic-array, "connection machine," or general-purpose equipment.

In applications of the memory, the words are patterns of features. Some features are produced by a sensory system, others control a motor system, and the rest have no immediate external significance. There is a *current* (1,000 bit) *pattern*, which is the current contents of the system's *focus*. The sensors feed into the focus, the motors are driven from the focus, and the memory is accessed through the focus. What goes on in the world—the system's "subjective" experience—is represented internally by a sequence of patterns in the focus. The memory stores this sequence and can recreate it later in the focus if addressed with a pattern similar to one encountered in the past. Thus, the memory learns to predict what is about to happen.

Wide applications of the memory would be in systems that deal with real-world information in real time. Such information is rich, varied, incomplete, unpredictable, messy, and dynamic. The memory will home on the regularities in the information and will base its decision on them. The applications include vision—detecting and identifying objects in a scene and anticipating subsequent scenes—robotics, signal detection and verification, and adaptive learning and control. On the theoretical side, the working of the memory may help us understand memory and learning in humans and animals.

For an example, the memory should work well in transcribing speech, with the training consisting of "listening" to a large corpus of spoken language. Two hard problems with natural speech are how to detect word boundaries and how to adjust to different speakers. The memory should be able to handle both. First, it stores sequences of patterns as pointer chains. In training—in listening to speech—it will build a probabilistic structure with the highest incidence of branching at word boundaries. In transcribing speech, these branching points are detected and tend to break the stream into segments that correspond to words. Second, the memory's sensitivity to similarity is its mechanism for adjusting to different speakers—and to the variations in the voice of the same speaker.

1.2 Rationale for special hardware

Although the sparse distributed memory is a generalized random-access memory, its most important properties are not demonstrated by ordinary random accesses. For those properties to appear, the memory addresses must be at least 100 bits and preferably several hundred, and any read or write operation must manipulate many memory locations. When these conditions are met, the memory can use approximate addresses (in the Hamming-distance sense) to retrieve exact information as well as statistically abstracted information that represents natural groupings of the input data. Intelligence in natural systems is founded on such properties.

Simulation of the memory on a conventional computer is extremely slow. A properly designed, highly parallel hardware is absolutely necessary for dealing with practical problems in real time. Table 1.1 shows the estimated performance of different sized memories on a range of hardware implementations.

The Stanford prototype is designed to be a flexible, low-cost model of projected large-scale implementations. Experiments performed with the prototype are intended to develop better applications support and especially faster, more efficient implementations.

Table 1.1: Realizing sparse distributed memory in different kinds of hardware.

Hardware	Dimension, n	Number of locations, m	Cycles per second	Task
Dedicated DEC 2060	128	10,000	.2–1	Demonstrate convergence properties of the memory
32-node Intel iPSC	128	50,000	1–5	Simple learning by trial and error
16K-processor Connection Machine	200	60,000	50–200	Word parsing in compacted text
Stanford Prototype	256	80,000	50	Word parsing in compacted text and possibly in speech
Present VLSI potential	1,000	100,000,000	1,000	Language understanding (?)

1.3 Basic concepts and terminology

This chapter presents a nonmathematical description of the operating principles behind SDM. Readers desiring a mathematical description of these concepts should consult the book by Kanerva [4]. The papers by Keeler [5] and Chou [1] contrast the properties of SDM with a neural-network model developed by Hopfield [3] that resembles SDM in certain aspects of its operation.

There are six concepts that are central to describing the behavior of SDM. These are:

- Writing to the memory.
- Reading from the memory.
- Address pattern (or reference address, or retrieval cue, or cue).
- Data pattern (or contents, or data word).
- Memory location (or hard location) and hard address.
- Distance from a memory location.

The first two are operations on the memory, the middle two are external to the memory and have to do with the external world, while the last two are concepts relating to the internal aspects of the memory. Each of these is explained in more detail below.

Writing is the operation of storing a data pattern into the memory using a particular address pattern.

Reading is the operation of retrieving a data pattern from the memory using a particular address pattern.

Address Pattern. An N-bit vector used in writing to and reading from the memory. The address pattern is a coded description of an environmental state. (In the prototype, $N = 256$.)

Data Pattern. An M-bit vector that is the object of the writing and reading operations. Like the address pattern, it is a coded description of an environmental state. (In the prototype, $M = 256$.)

Memory location. SDM is designed to cope with address patterns that span an enormous address space. For example, with $N = 256$ the input address space is 2^{256}. SDM assumes that the address patterns actually describing physical situations of interest are sparsely scattered throughout the input space. It is impossible to reserve a separate physical location corresponding to each possible input; SDM implements only a limited number of physical or "hard"' locations. The physical location is called a *memory* (or *hard*) *location.*

Every hard location has associated with it two items:

- A fixed *hard address*, which is the N-bit address of the location.
- A *contents* portion that is M-bits wide and that can accumulate multiple M-bit data patterns written into the location. The contents' portion is not fixed; it is modified by data patterns written into the memory.

1.3.1 Distance from a memory location (to the reference address)

The distance from a memory location to a reference address used in either a read or write operation is the Hamming distance between the memory location's hard address and the reference address. The Hamming distance between two N-bit vectors is the number of bit positions in which the two *differ*, and can range from 0 to N. SDM uses the Hamming measure for distance because it is convenient for vectors consisting of 0s and 1s. However, other measures could equally well be used.

The operation of the memory is explained in the remainder of this chapter. However, the following is a brief preview:

- During a write, the input to the memory consists of an address pattern and a data pattern. The address pattern is used to select hard locations whose hard addresses are within a certain cutoff distance from the address pattern. The data pattern is stored into each of the selected locations.
- During a read, an address pattern is used to select a certain number of hard locations (just like during a write). The contents of the selected locations are

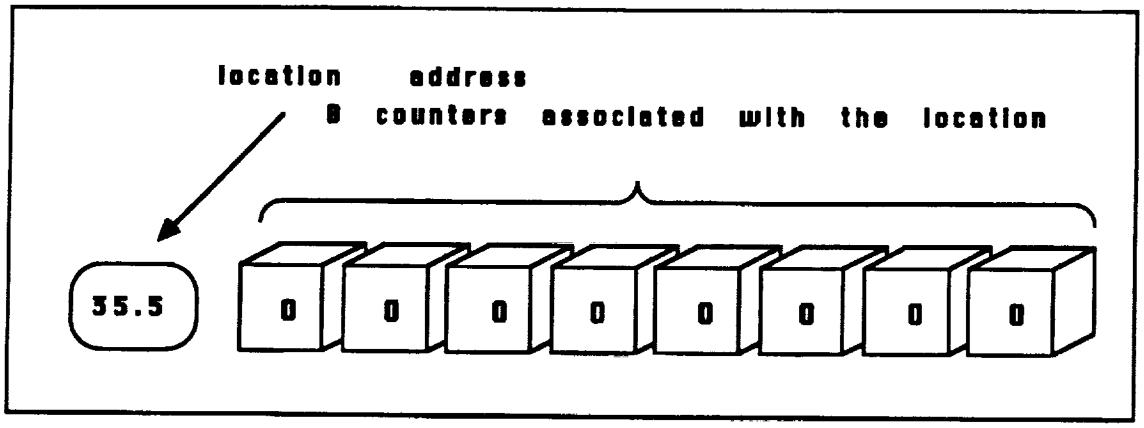


Figure 1.1: Example of a location.

bitwise summed and thresholded to derive an M-bit data pattern. This serves as the output read from the memory.

How this works is explained in the following section.

1.4 Basic concepts

1.4.1 A simple example

These concepts and the basic mechanisms of SDM will be illustrated by a stylized example. For the sake of simplicity, assume the following:

1. Input vectors consist of:

   (a) an integer address that can range from 1 to 1000, and

   (b) a data pattern (or content portion) that is an 8-bit vector.

   An example of an input vector is: *Address* *Data pattern*
   867 0 1 1 0 1 0 1 0

   It should be emphasized that the data pattern is not a binary number. Rather, the 1s and 0s could be thought of as the presence or absence of specific features. In the actual implementation, described later, both the address and contents are 256-bit patterns.

2. The memory in this example implements only 99 hard locations. These have associated with them the addresses:

   5.5, 15.5, 25.5, . . . , 995.5

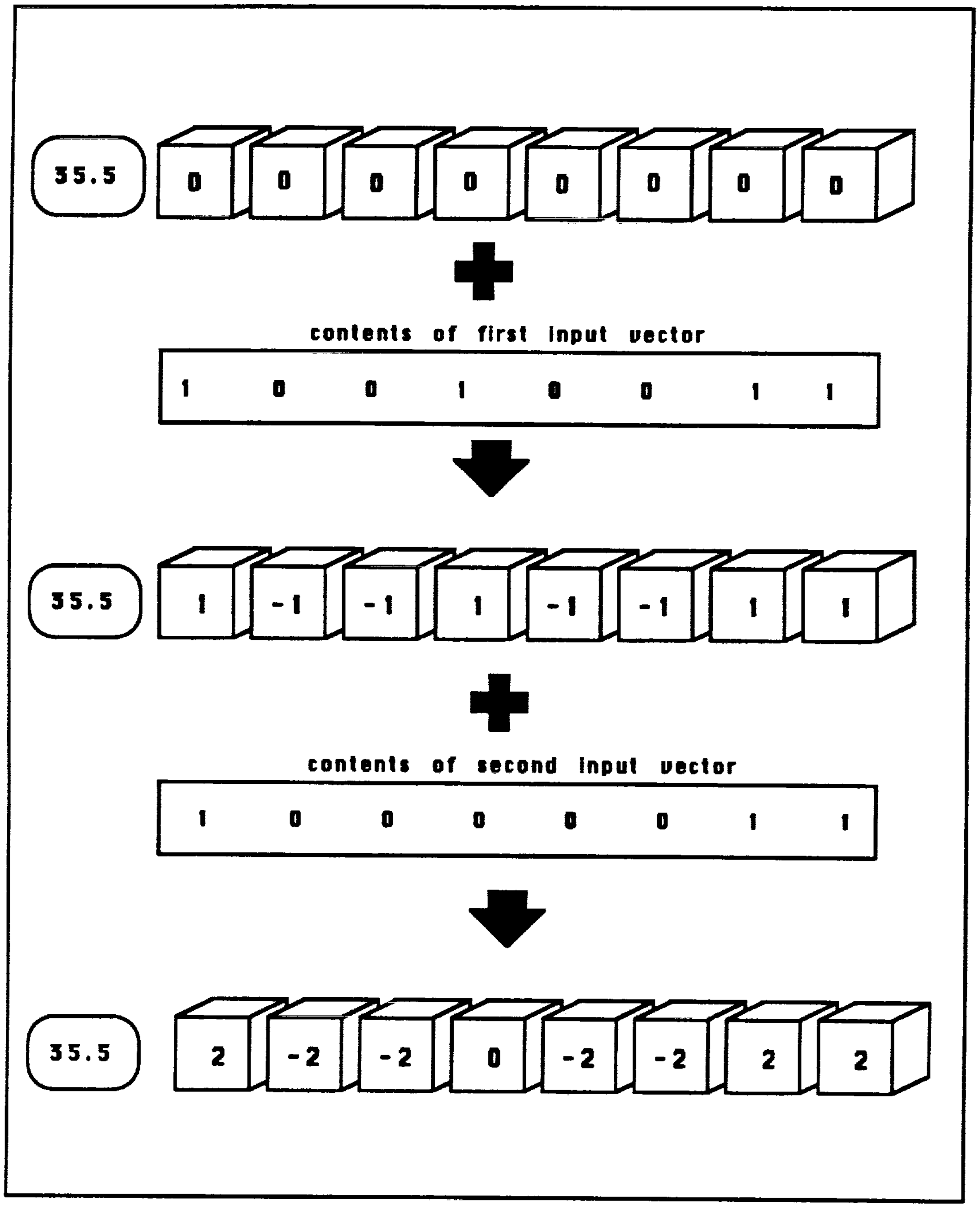


Figure 1.2: Example of a location.

The reason for the half addresses is merely to position each location symmetrically between 1 and 10. The need for this will be clear shortly.

3. Each hard location has associated with it 8 buckets—one bucket for each bit of an 8-bit data-pattern vector. Each bucket accumulates bits that are stored into it by acting as an up/down counter. Each bucket starts out holding the value 0. A binary 1 stored into the bucket causes its count to go up by 1, whereas a binary 0 stored into a bucket causes its count to go down by 1.

   As will be explained shortly, this facility is required because each location may have many inputs stored into it.

An example of a location is shown in Figure 1.1. If an input vector with contents 1 0 0 1 0 0 1 1 is stored into this location, the location will look as shown in the upper half of Figure 1.2. If now another input vector with contents 1 0 0 0 0 0 1 1 is stored into the same location, the result is shown in the lower half of that figure.

The contents of an individual location could be interpreted as follows. If a bucket has a count that is positive, it has had more 1s written into it than 0s and can be interpreted as a 1. Similarly, a bucket with a negative count can be interpreted as a 0. A bucket with a 0 count (in a location that has been written into) has had an equal number of 1s and 0s written into it and can be interpreted as a 1 or a 0, each with probability 0.5.

To understand the working of SDM, we will deal with the problem of retrieving the closest match. We want to store into memory the input vectors that the system encounters. At some later point, we want to present to the memory an address cue and have the memory retrieve the contents of the stored input vector with the address that is closest to the input cue.

1.4.2 A simple solution that does not work

An apparently simple way in which this best-match problem could be tackled is the following:

> Store each input into the closest hard location. This can be accomplished by making each hard location "sensitive" or addressable by any input with an address that is within 4.5 of the address of the location. Thus, any input with an address in the range 31 to 40 (both inclusive) would be stored into the location with the address of 35.5, and when presented with a retrieval cue would read out the contents of the closest hard location.

Unfortunately, though this sometimes works, it often does not. To understand this, consider the following example:

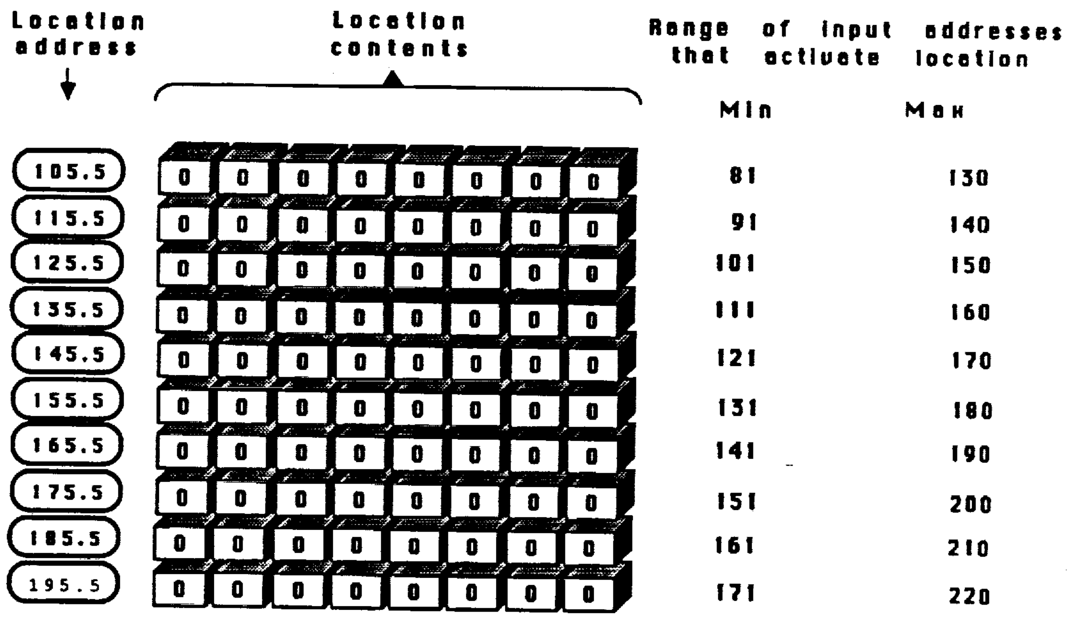


Figure 1.3: Example using SDM.

Input #1:	139	1 0 1 0 1 0 1 0
Input #2:	169	1 1 0 0 1 0 1 1
Retrieval cue:	150	

Input #1 will be stored into the location with address 135.5. Input #2 will be stored into the location with address 165.5. The retrieval cue will activate the location 155.5. This location has nothing in it. One way to deal with this problem is to gradually increase the activation distance during retrieval. In the above case, if the activation distance were increased to $\pm$ 14.5, the system would retrieve the contents of the location 135.5, which contains the closest match. However, if the example is modified slightly so that the first input address is 131 and the second is 161, the method fails even after the activation range has been increased to 150 $\pm$ 14.5. SDM solves the problem using a statistical approach that is much more robust and has fairly simple mechanics.

1.4.3 The SDM solution

SDM overcomes the above problem by:

1. Distributing each stored input over many locations, and

2. Retrieving from a distributed set of locations.

This is the reason for the word "distributed" in the name of the system. Now, instead of storing an input into the closest location, an input is stored into all locations within a certain distance of the write address. Similarly, when presented with a retrieval cue, all locations within a certain distance of the retrieval cue are read out and used to derive the output in a manner to be described. These two distances, namely the activation distances during the storage and retrieval of patterns, need not be the same. The operation of SDM can be illustrated by continuing with the previous example. Instead of having each physical location be addressable by any address within 4.5 of it, assume that the activation distance is now ±25. We will use the same activation distance during both the storage and retrieval phases, for simplicity. Figure 1.3 illustrates the initial state of a portion of the system, encompassing physical locations with addresses ranging from 105.5 to 195.5. Also shown is the range of addresses to which each location is sensitive.

When the memory is presented with the first input pattern, 139: 1 0 1 0 1 0 1 0, the memory locations with addresses 115.5, 125.5, 135.5, 145.5, and 155.5 are all activated. The contents of the input vector are written into each of the locations according to the rule described earlier. Figure 1.4 shows the state of the system after this occurs.

Now the system is presented with Input #2, namely 169: 1 1 0 0 1 0 1 1. This input activates the locations with addresses 145.5, 155.5, 165.5, 175.5, and 185.5. The vector being stored, 1 1 0 0 1 0 1 1, is accumulated bit by bit into the buckets of each of these locations. The resulting state of the system is presented in Figure 1.5. Notice that the two locations at addresses 145.5 and 155.5 have each had both input vectors written into them. Both input vectors fell within the ±25 activation distance of each of these locations.

Now consider what happens when the system is presented with the retrieval cue 150. This address activates all locations with addresses in the range 150 ± 25, namely, the locations with addresses 125.5, 135.5, 145.5, 155.5, and 165.5. The retrieval mechanism's goal is to determine, for each bit position, whether more 1s or more 0s were written into all the selected locations and to output 1 or 0 accordingly. In the case of a tie, 1 or 0 is output with probability 0.5.

The way this works is illustrated in Figure 1.6. For *each bit position,* the following operations are performed:

1. The contents of the buckets of all the selected locations are summed arithmetically. (A positive sum indicates that more 1s were written into these locations, while a negative sum indicates more 0s.)

2. The sum is thresholded.

A positive sum yields 1, a negative sum yields 0, and a sum of 0 yields 1 or 0 based on the toss of a coin. In this particular case, this process yields the output 1 0 1 0 1 0 1 0. This is the system's response to the query "What is the content portion of

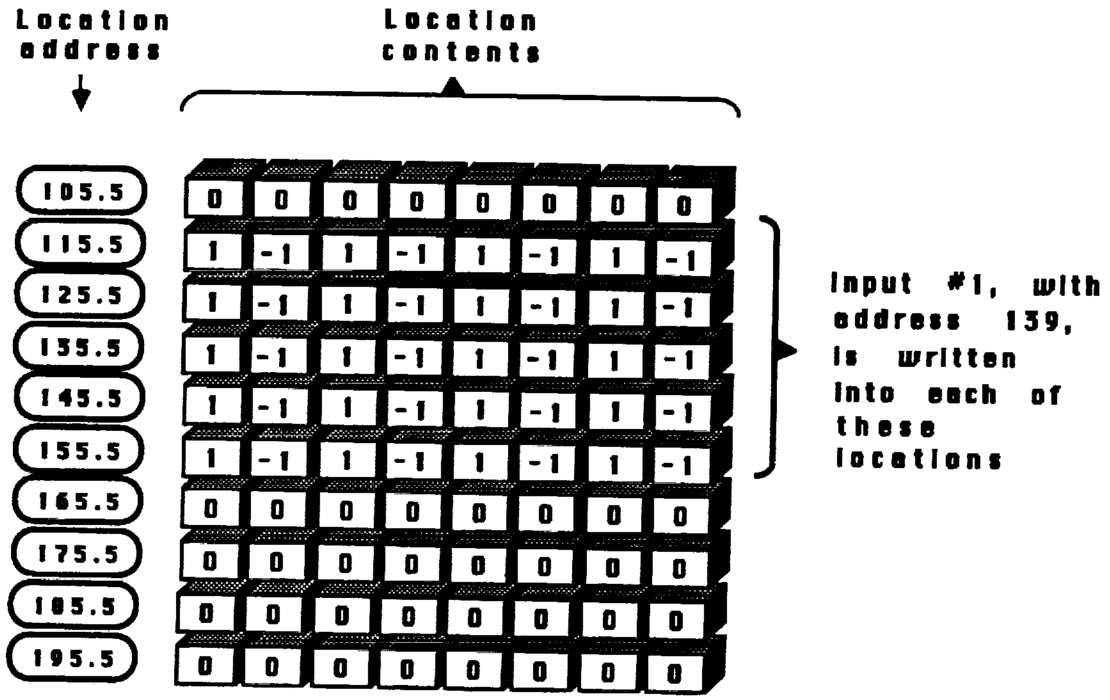


Figure 1.4: Example using SDM.

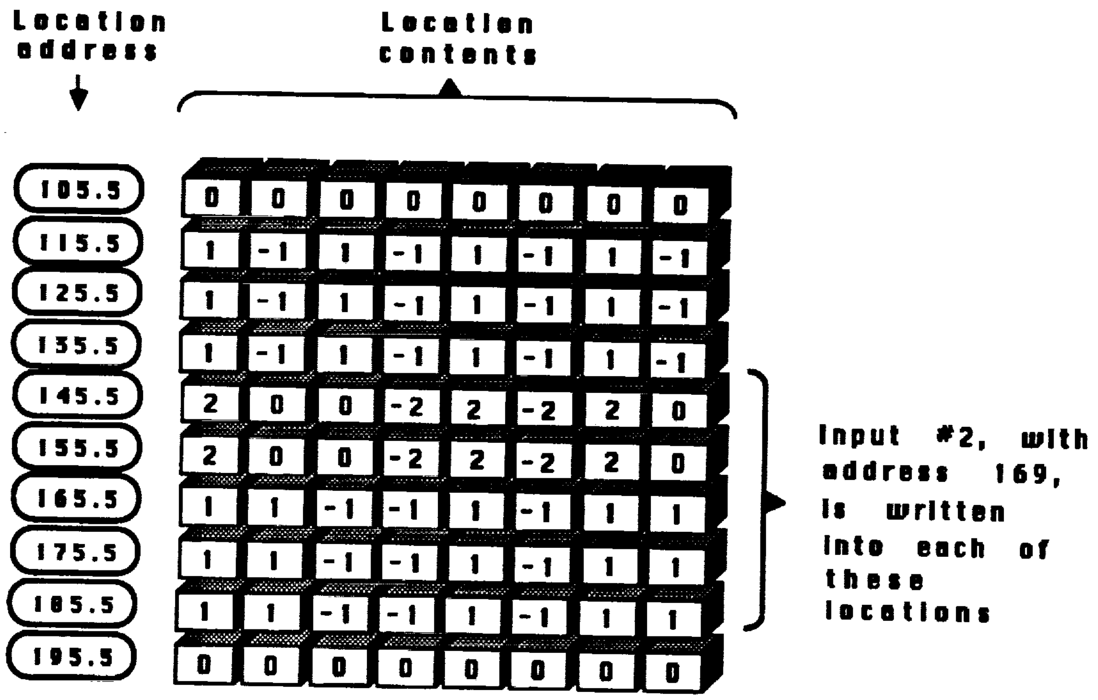


Figure 1.5: Example using SDM.

the stored input with the address closest to the retrieval cue of 150?". Notice that the vector output by the system is in fact the content portion of Input #1, which was the stored input vector with the closest address to 150, the retrieval cue.

1.4.4 Differences between the simple example and the real model

The simplified model presented above is sufficient to explain some of the basic mechanisms of SDM. However, to understand the workings of the model in more depth we switch now to a discussion based on the real model. In the simple model above, input vectors consisted of an address portion that was a three digit integer, while the contents were an 8-bit vector. Distances between input vectors were measured by the magnitude of the arithmetic difference between addresses. In the actual model, input vectors have an address consisting of an N-bit vector and a contents portion consisting of an M-bit vector. N and M need not be the same in the most general case, but they are both 256 in the prototype (see note on page 14). This equality is required for certain modes of operation, described later in this report. Distances between vectors are measured by the Hamming distance between the N-bit vectors. Since the Hamming distance is the number of bit positions in which the two vectors differ, for N-bit vectors this distance will range from 0, for two identical addresses, to N, for two addresses that are bitwise complements. The potential address space is 2^{256}, compared to 1,000 in the simple example. Whereas the simple model had 100 hard locations, the basic module of the prototype system has 8,192 locations with addresses scattered randomly with uniform probability through the address space. Each hard location has associated with it a set of 256 buckets to accumulate the vectors that may be stored into that location. Each bucket is conceptually an 8-bit up/down counter that can hold a count in the range of -127 to $+127$, inclusive. (If more than 127 1s are stored into a bucket in excess of 0s, the bucket will saturate at 127. Similarly, it will saturate at -127 if the number of 0s stored into a bucket exceeds the number of 1s by more than 127.) For the discussion that follows, it is useful to visualize the 2^N input space as a two-dimensional rectangle. Figure 1.7 shows the address space in this fashion. The physical locations are indicated by the small black dots within the rectangle.

The process of writing or storing a vector into the memory consists of the following two steps:

1. Draw an N-dimensional sphere of Hamming radius d around the address of the input vector. In the plane this can be visualized as drawing a circle centered at the address of the input vector.

2. For each physical location that falls within this sphere, accumulate the contents portion of the input vector into each of the 256 associated buckets. This is depicted in Figure 1.8.

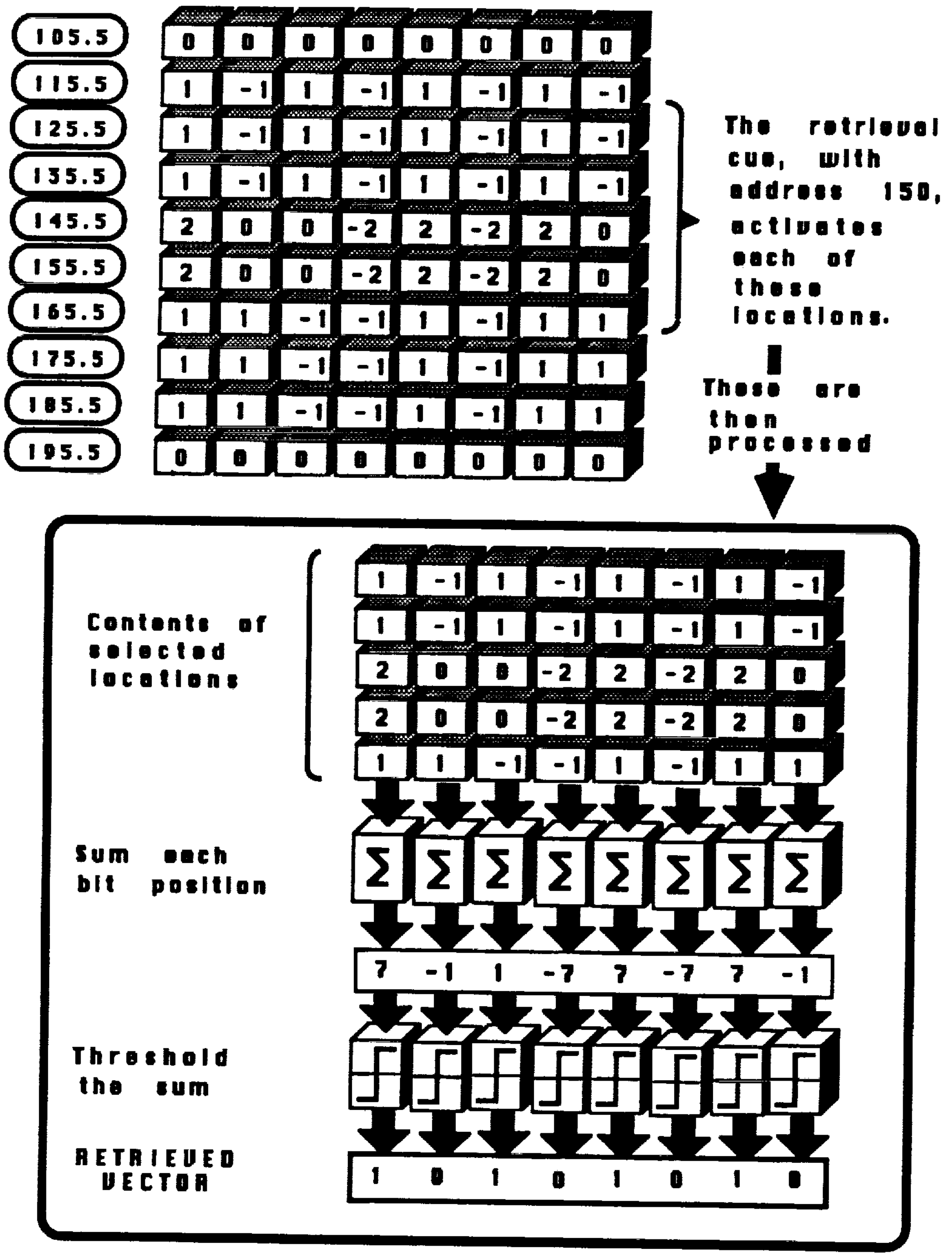


Figure 1.6: Example of data selection in SDM.

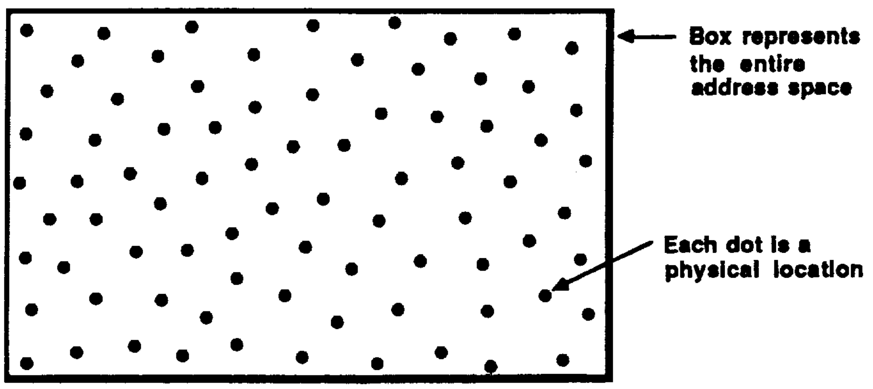

Box represents the entire address space
Each dot is a physical location

Figure 1.7: Space of locations.

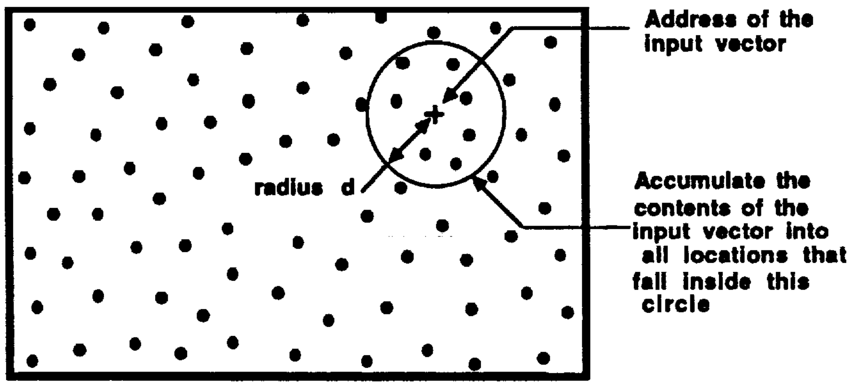

Address of the input vector
radius d
Accumulate the contents of the input vector into all locations that fall inside this circle

Figure 1.8: SDM storage activation radius.

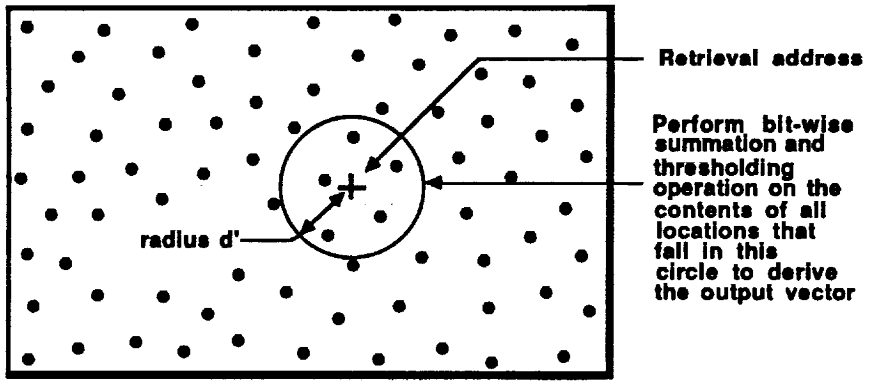

Retrieval address
radius d'
Perform bit-wise summation and thresholding operation on the contents of all locations that fall in this circle to derive the output vector

Figure 1.9: SDM retrieval activation radius.

Given a retrieval address, the process of reading from the memory proceeds in a similar two-step fashion:

1. Draw an N-dimensional sphere of Hamming radius d' (which need not equal the radius d used for storing patterns) centered at the retrieval cue.
2. Derive the i^{th} bit of the output vector (i going from 1 to M) in a manner identical to that used in the simple example.

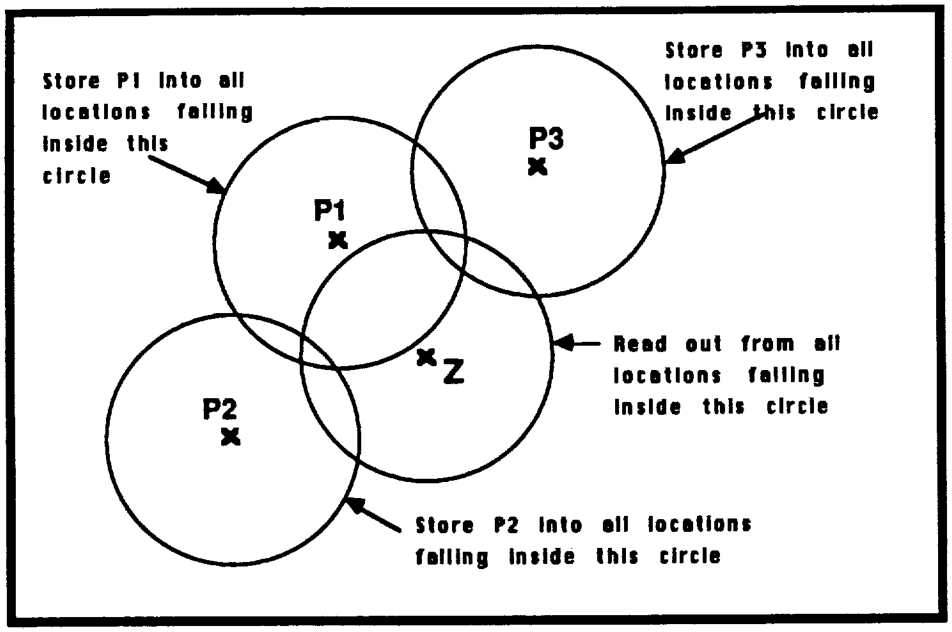
Specifically, sum the contents of the i^{th} buckets of all the locations falling within the sphere drawn above, and then threshold the sum to either 1 or 0 based on whether the sum is positive or negative. This is depicted in Figure 1.9.

Note: The software could in theory be modified so instead of having $m = 256$ 8-bit counters associated with each hard location, one could have $m = 128$ 16-bit counters, or $m = 64$ 32-bit counters. The positive and negative ceilings for the counters would then also be changed in software to ± 32767 and ± 2147483647, respectively. Since the microprocessor on the stack module uses a 32-bit word-length, it is impractical to implement counters larger than this.

1.5 Autoassociative dynamics

We are now in a position to understand the reconstructive properties of SDM when used in the autoassociative mode. In this mode, the dimensionality of the address and contents portions of an input vector are the same. In fact, the contents are also used as the address. This means that the input vector can be viewed as consisting of a pattern vector P that forms its address vector, and the same pattern vector P that forms it contents vector. During storage, the pattern vector P serves as the address that is used to select a number of physical locations. The same vector P is also stored into each of the locations it activates. Figure 1.10 shows three pattern vectors $P(1)$, $P(2)$, and $P(3)$ stored into locations in memory. Z is a contaminated or partial version of $P(1)$, and the goal is to have the memory reconstruct $P(1)$ when cued with Z. As shown in the diagram, the locations activated by Z are of six types:

1. Those that contain only $P(1)$.
2. Those that contain $P(1)$ and $P(2)$.
3. Those that contain $P(1)$ and $P(3)$.
4. Those that contain only $P(2)$.
5. Those that contain only $P(3)$.


Store P1 into all locations falling inside this circle
Store P3 into all locations falling inside this circle
P3
P1
Z
P2
Read out from all locations falling inside this circle
Store P2 into all locations falling inside this circle

For clarity, individual physical locations are not shown.

Figure 1.10: Overlapping radii.

6. Those that contain nothing.

This can be generalized to say that the locations activated by Z contain a mixture of $P(1)$, which can be regarded as the signal, and non-$P(1)$ patterns, which can be regarded as noise, using nomenclature from Keeler [5].

The signal-to-noise ratio is higher (a) the closer Z is to $P(1)$, and (b) the less densely populated the memory is, i.e., the fewer the patterns that have been written into it. It can be shown mathematically [4] that if Z is closer than a certain critical distance to $P(1)$, then summing and thresholding the contents of the locations activated by Z results in a vector $Z(1)$ that is closer to $P(1)$ than Z was. The critical distance is a function of how densely populated the memory is. The greater the population of the memory, the smaller the critical distance. The fact that $Z(1)$ is closer to $P(1)$ than Z was is of great benefit because now $Z(1)$ can be used as a new retrieval cue. The output $Z(2)$ will be even closer to $P(1)$ than $Z(1)$ was, and it too can now be used as a new retrieval cue. This iterative process, which is a form of feedback, therefore produces a sequence of outputs $Z(1)$, $Z(2)$, $\ldots$, $Z(n)$ that converges rapidly to either $P(1)$ or a minimally noisy version of $P(1)$. This is depicted in Figure 1.11a. If the cue vector Z is beyond the critical distance from $P(1)$, i.e., if it is too contaminated or incomplete, the sequence of vectors $Z(1)$, $Z(2)$, $\ldots$ will not converge to $P(1)$. Instead, it will be a diverging sequence that will wander through address space. It may eventually wander, by chance, into the attracting zone of some other stored pattern, say, $P(k)$, and thereby converge onto $P(k)$. This process is depicted in Figure 1.11b.

Experimentally, convergence to the correct value $P(1)$ occurs rapidly ($\sim$10 iterations), whereas the diverging sequence, if it eventually converges to some other pattern, takes a large number of iterations. The two situations are easy to distinguish in practice.

There is an upper bound to the value of the critical distance referred to above. In a sparsely populated memory in which there is little or no overlap between the locations in which patterns have been stored, the critical distance is necessarily less than the sum of the Hamming radii used during storage and retrieval. If the cue Z is beyond this distance from $P(1)$, none of the locations activated by it during the retrieval process will contain any copies of $P(1)$ (see Kanerva [4] for details).

The behavior of SDM illustrated in Figure 1.11 is similar to the dynamic behavior of the Hopfield net used in its autoassociative mode. (See Keeler [5]). In that model, stored patterns act like attractors for input cues within a certain critical distance. The behavior of the Hopfield model, however, is driven by an energy-minimization mechanism in which the stored patterns behave like local minima of an energy function associated with the address space.

1.6 Heteroassociative dynamics (sequences)

SDM can also be used in a heteroassociative mode. In this mode the contents portion of an input vector is not generally equal to its address portion. In general, the two do not have to be of the same dimension either. However, when they do have the same dimensionality, SDM lends itself to the storing and recalling of sequences. Consider a sequence of N-dimensional pattern vectors:

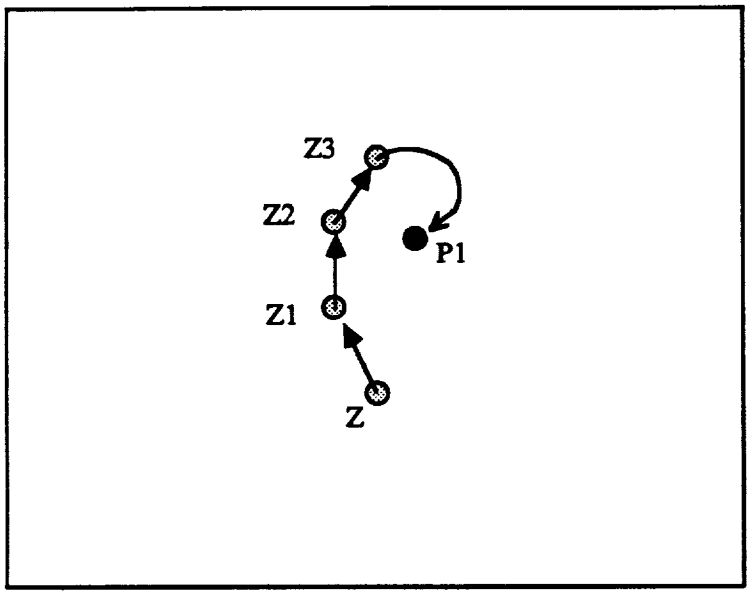
$$P(1), P(2), P(3), \ldots$$

Examples of such sequences might be (a) a sequence of motor-control parameters for controlling the trajectory of a robot arm, or (b) a sequence of musical notes that comprises a tune.

Imagine forming the input vectors shown below, where the address portion of an input vector is the previous data pattern element in the sequence.

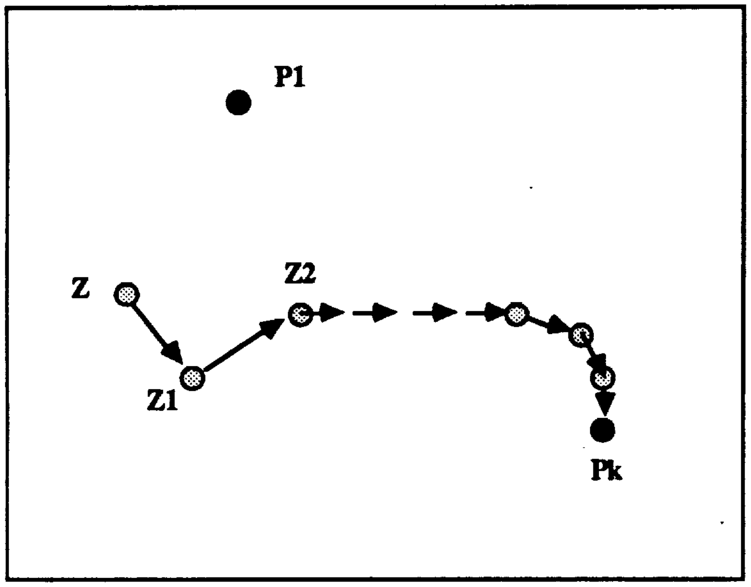
Address pattern	*Data pattern*
$P(1)$	$P(2)$
$P(2)$	$P(3)$
$P(3)$	P(4)
$\vdots$	$\vdots$
$P(i)$	$P(i+1)$

These vectors can be used to write into the memory. The effect of this is that at the locations selected by the address $P(1)$, the pattern $P(2)$ is stored; at the locations



(a)

Cueing the memory with Z sufficiently close to P1 results in a rapidly converging seqence of values. The stored pattern P1 acts like an attractor.



(b)

If Z is too distant from P1, the iteratively retrieved sequence does not converge. It may eventually converge on some other pattern Pk.

Figure 1.11: SDM autoassociative mode.

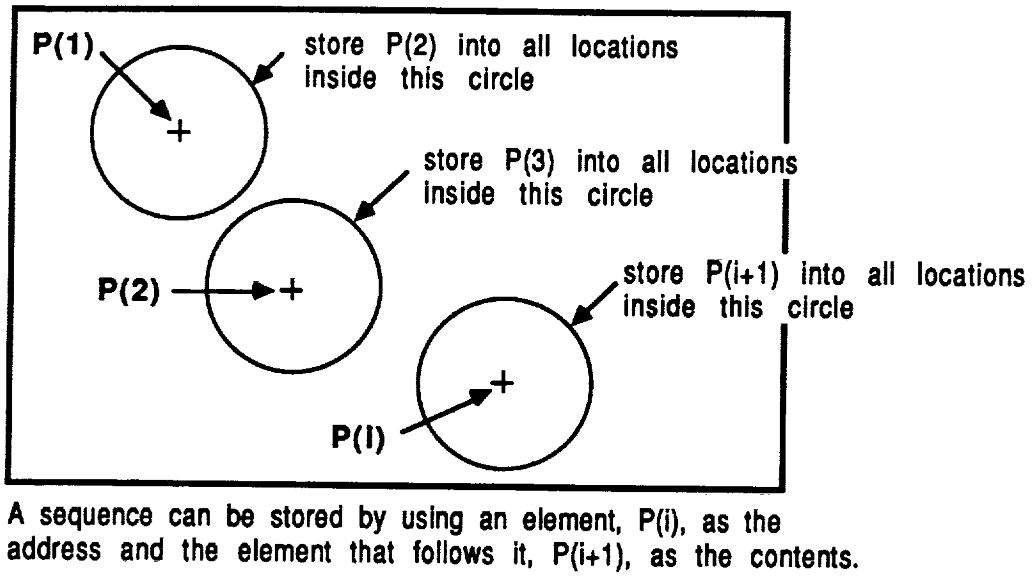
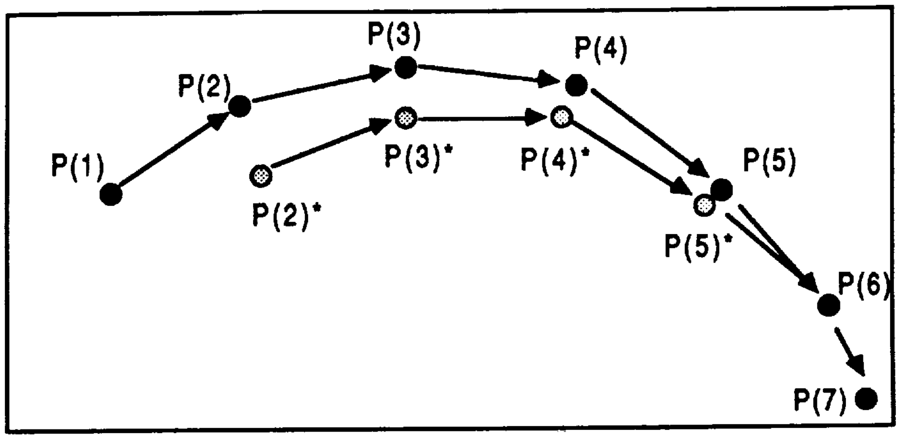

P(1)
store P(2) into all locations inside this circle
store P(3) into all locations inside this circle
P(2)
store P(i+1) into all locations inside this circle
P(i)
A sequence can be stored by using an element, P(i), as the address and the element that follows it, P(i+1), as the contents.

Figure 1.12: SDM heteroassociative mode: storage.

selected by the address $P(2)$, the pattern $P(3)$ is stored, and so on. In general, at the locations selected by the address $P(i)$, the pattern $P(i+1)$ is stored. This is illustrated in Figure 1.12.

Now imagine cueing the memory with a pattern close to one of the elements of the stored sequence. For example, suppose we cued the system with the address $P(2)*$ that is close to $P(2)$. Just as in the autoassociative case, if $P(2)*$ is sufficiently close to $P(2)$, the retrieved pattern will be even closer to the pattern that was stored in the locations activated by the address $P(2)$. In this case the retrieved value $P(3)*$ will be closer to the stored value $P(3)$ than $P(2)*$ was to $P(2)$. If $P(3)*$ is now used to cue the memory, the retrieved pattern P(4)* will be closer to P(4) than $P(3)*$ was to $P(3)$. Continuing in this manner, we observe that cueing the memory with the pattern $P(2)*$ allowed us to iteratively recover the sequence: $P(3)*$, $P(4)*$, $P(5)*$, ... that converges onto the stored sequence $P(3)$, $P(4)$, $P(5)$, ... This is illustrated in Figure 1.13.

Just as in the autoassociative case, if the initial cue $P(2)*$ is too distant from $P(2)$, the retrieved sequence would not converge to the stored sequence. However, unlike the autoassociative case where convergence can be easily distinguished from divergence, in the case of sequences the difference is unfortunately hard to tell by looking at the retrieved patterns.



Using a cue close to any member of a stored sequence iteratively recovers a sequence that converges to the stored sequence.

Figure 1.13: SDM heteroassociative mode: retrieval.

1.6.1 Folds

The ability to store sequences endows SDM with the capability to behave as a predictor. The values recovered from stored sequences provide a prediction of the most probable future event. This is illustrated in the following example. Suppose the sequence $A \rightarrow B \rightarrow C \rightarrow D$ occurs more often than the sequence $A \rightarrow B \rightarrow E \rightarrow D$. Suppose, further, that each sequence that the system encounters is written into it in the manner previously described. If the system now encounters $B*$ (close to B), what is likely to happen next? Cueing the system with $B*$ will recover $C*$ (close to C) rather than $E*$ (close to E), because in the locations activated by B there were more copies of C stored than of E, simply because it occurred more often. Thus, the retrieval mechanism predicts the most likely next step in the sequence.

The examples used so far have associated the next element in a sequence with the one before it. This is often insufficient as a basis for prediction. For example, consider the two equiprobable sequences:

$$A \rightarrow B \rightarrow C \rightarrow D$$

$$E \rightarrow B \rightarrow C \rightarrow F$$

Given an event $C*$, we have insufficient information to predict the next event. In fact, to do so we need to look not only one but two steps back in the sequence to know which sequence we are in. SDM handles such situations by utilizing "folds" of different "orders" and combining the results from different folds to arrive at the result. In general, a k^{th}-order fold is a complete set of SDM locations in which sequences are stored with pattern $P(i)$ serving as the address and pattern $P(i+k)$ serving as the contents. More specifically, a first-order fold is one in which the pattern stored is the one that immediately follows the pattern that forms the address. A second-order fold is one in which the stored pattern is the one that follows the address pattern by two steps, and in a third-order fold the stored pattern follows the address pattern by three steps. The two sequences listed above would result in the following storage in each of three folds:

Sequence	$A \to B \to C \to D$	$E \to B \to C \to F$
1st-order fold	at A store B at B store C at C store D	at E store B at B store C at C store F
2nd-order fold	at A store C at B store D	at E store C at B store F
3rd-order fold	at A store D	at E store F

Figure 1.14 shows how to use multiple folds to arrive at a prediction. Imagine that the system has previously encountered the sequences $A \to B \to C \to D$ and $E \to B \to C \to F$ in an equiprobable way and that it has stored patterns into its three folds in the manner shown above. Now, assume that the system encounters the patterns E, B, and C in that order. The input being encountered is fed into a mechanism like a shift register in which each register holds a pattern. The shift-register contents are used as input cues to successively higher order folds. In this case, the most recent input pattern C is used as an input cue to the first-order fold, B is used as a cue to the second-order fold, and E is used as a cue to the third-order fold. To derive a result, the standard summing and thresholding operation is performed on the contents of *all* the locations activated, not fold by fold. The locations activated by the cue C in the first-order fold have had an equal number of D and F patterns written into them, as have the locations activated by the cue B in the second-order fold. The cue E activates locations in the third-order fold that have only had F patterns written into them. The result of summing all of these together and then thresholding is that the pattern F is recovered with high probability. This pattern is the system's prediction of the next event that is likely to occur.

A major difference between the model and the prototype lies in the representation of the data word. In the model, the input vector (referred to as a "word" from here on) is an m-bit binary vector of 1s and 0s, while in the prototype it is an m-byte vector where each byte encodes the value of one vector component. Moreover, binary 1s are coded as $+1$, and binary 0s are coded as -1. This representation provides greater flexibility in application settings.

Some of the parameters of the prototype design were described in section 1.4.4 on page 11. Input vectors are 256-word-long vectors. The system implements 8,192 hard addresses (locations). Words written into the system are accumulated, bit wise, into

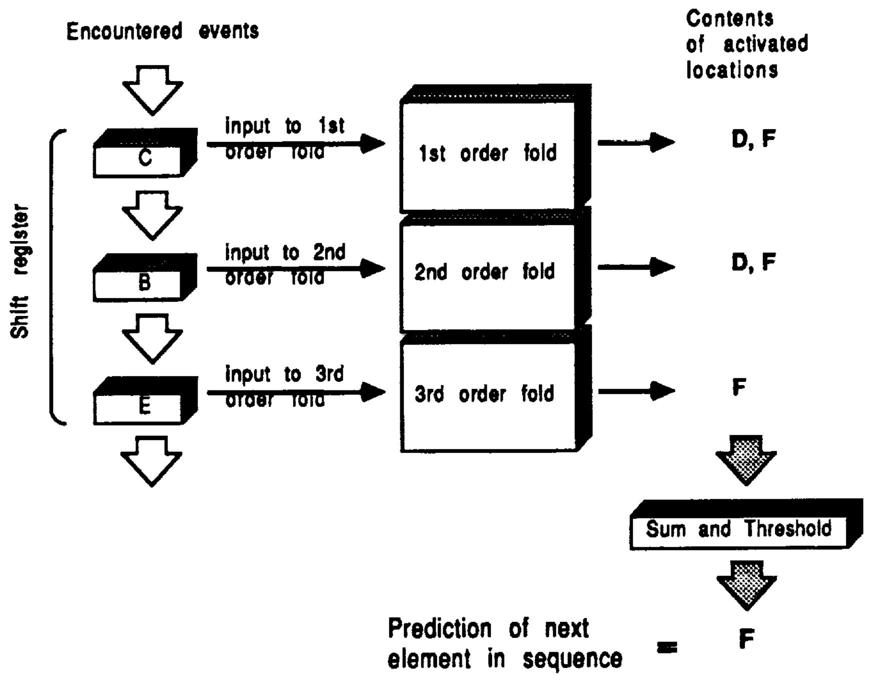


Figure 1.14: Folds.

Table 1.2: Parameters of SDM prototype.

Dimensionality of a word:	256
Counter size:	16 bits
No. of Hard Locations:	8192
Number of folds:	1 to 16
Hamming Radius:	0 to 255
Number of reads or writes per second:	50

Table 1.3: Hardware and software components for prototype system.

Module	Underlying Hardware	Underlying Software
Executive module	Sun 3/60 workstation, or any microcomputer with SCSI port	Custom C-code
Control module	MC68030 based single-board computer	Assembly language code
Address module	Custom design using LSI components on wire-wrap board	None
Stack module	MC68030 based single-board computer (1 per fold)	Assembly language code

buckets that hold an 16-bit binary count. These and some performance characteristics of the system are summarized in Table 1.2.

The prototype system was designed around four modules. This modular approach provides the flexibility to modify memory parameters for the present project and makes it easy to upgrade specific portions of the system in future designs. Figure 1.15 shows a physical and block diagram of the system, while Table 1.3 shows the hardware and software used to implement each module.

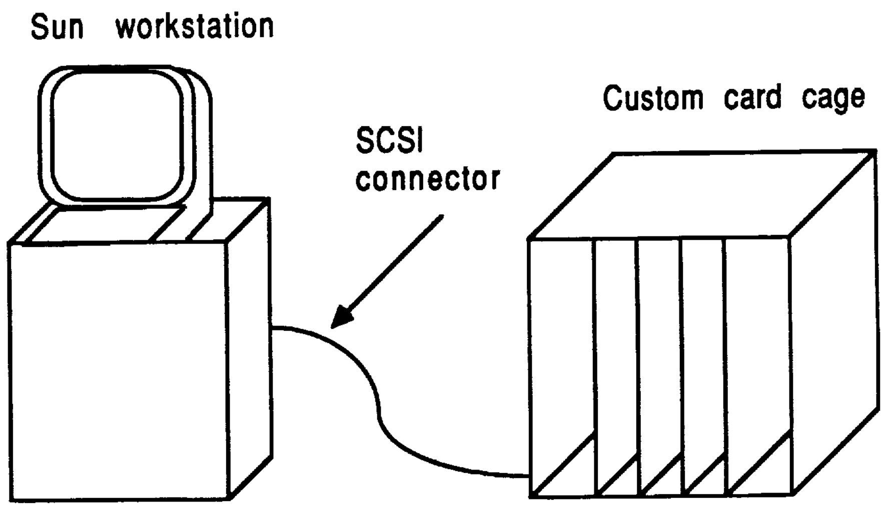
Each of these modules is described briefly below, and the description is expanded upon later in this chapter.

1.7 Physical description

The Executive module is a software module on the Sun 3/60 workstation that provides the interface between a user or application and the rest of the system. The system's "focus," described in section 1.1, resides here. The EM communicates with the rest of the system via a Small Computer Systems Interface (SCSI) port on the Sun. Executive module software is written in C.

The remaining modules in the system reside in a custom card cage, linked to each other by a VME bus.

The Control module controls the operation of the Address and Stack modules and acts as a link between them and the Executive module. It is connected to the EM on the Sun via the SCSI bus. The CM is implemented on a single-board microcomputer based on a Motorola MC68030 microprocessor. The board has 4 MB of random-access memory. The operating program is written in assembly language.



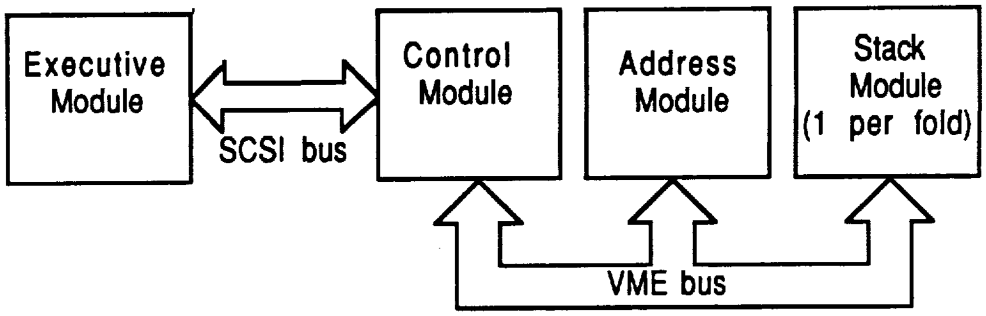


Figure 1.15: Physical and block diagrams of SDM prototype.

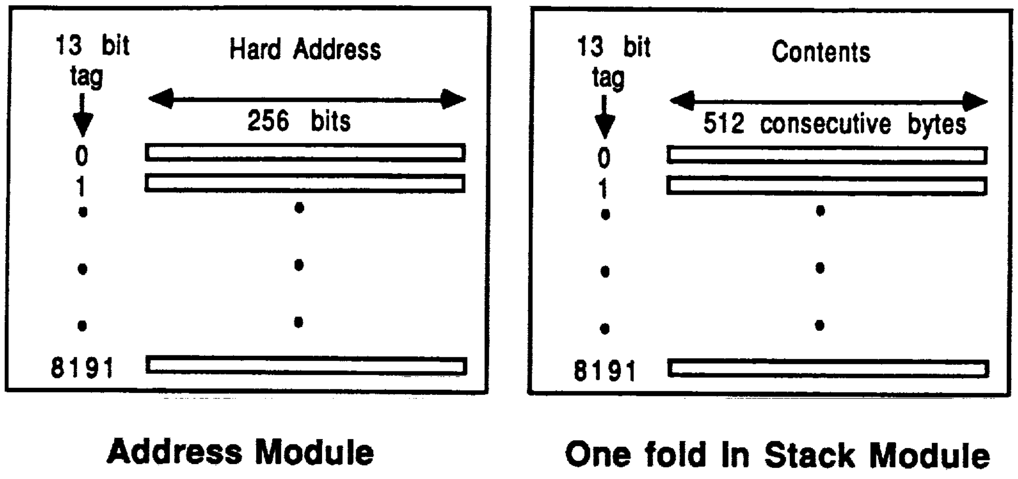


Figure 1.16: Relation between the Address and Stack modules.

The Address module performs the task of determining which hard addresses are within the specified Hamming distance of the reference address. Because of the computational intensity of this task, the Address module is custom designed and implemented on a wire-wrap board. The Address module is the only custom designed piece of hardware in the entire system.

The Stack module holds the contents of the folds. Each fold is implemented on a Force Computers MC68030-based single-board computer with 4MB of random-access memory. Each counter is implemented as 2 bytes (16 bits) in the memory space of the microprocessor, with the task of writing into the counter or reading from the counter being performed by the processor. Since the contents of a location consist of 256 counters, we use 512 sequential bytes to implement the 256 counters associated with each location.

1.8 Functional description

1.8.1 How the Address and Stack modules implement SDM locations: the concept of "Tags"

In a sparse distributed memory, each hard address has associated with it a set of counters in which to store words that are written into that location. In our implementation, the hard addresses are stored on one board (the Address module) while the counters associated with them are stored on a separate board (the Stack module). The one-to-one association is maintained via a 13-bit "tag" that associates a particular hard address on the Address module with a particular set of 256 counters on the Stack module. The conceptual arrangement is shown in Figure 1.16.

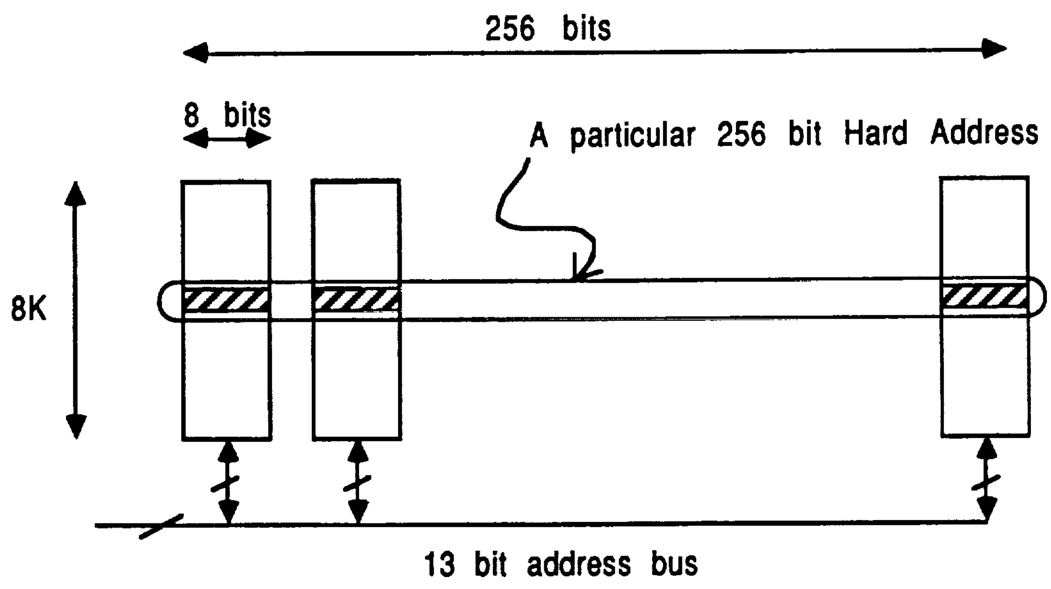


Figure 1.17: Physical arrangement of hard addresses in Address module.

Thus, when the Address module determines that a particular hard address is within the cut-off Hamming distance of the reference address, it simply passes the 13-bit tag associated with that hard address to the Control module. The Control module in turn passes the tag to the Stack module, which uses the tag to uniquely identify a set of 512 bytes that hold the contents of the location associated with the hard address in question. Physically, the Address module stores the hard addresses in a set of 32 static RAMs, each of which is 8K x 8 bits. Since these RAMs are operated in parallel, they behave like a set of locations that is 256 bits wide and 8192 deep, which is what we need to implement 8192 hard addresses. The arrangement is shown in Figure 1.17.

As described earlier, the Stack module implements the 256 counters associated with a location with 512 consecutive bytes in the address space of the MC68030 processor. Thus, the counter values associated with the hard address with tag 0 would be stored in bytes 0 to 511. For an arbitrary tag N, $(0 \leq N \leq 8191)$, the associated bytes would be at addresses $(N * 512)$ to $(N * 512 + 511)$. (The exact mapping is conceptually identical, though the locations in memory space are a little different).

1.9 Operational description

In order to explain how the prototype works, the description is divided into a set-up and operating phase. Each of these is explained in the following sections.

1.9.1 Set-up

The set-up phase consists of loading a set of up to 8192 hard addresses into the Address module. The user determines what hard addresses to use for a particular application. The Executive module passes each hard address and an associated tag to the Control module, which in turn passes them to the Address module, which writes the hard address into the physical location identified by the tag. (If one were starting a fresh application, there would be no reason to not load hard addresses into sequential locations in the Address module, and passing the tag from the Executive module would be redundant. However, this ability to load a hard address at a particular tag location is useful for testing performance characteristics and for debugging. For example, it allows one to change a particular hard address by simply storing a new one into the same tag location.)

1.9.2 Operation

During its operating mode, the SDM system either reads or writes. How each module works to achieve this is explained below, first for a write and then for a read.

SDM Write

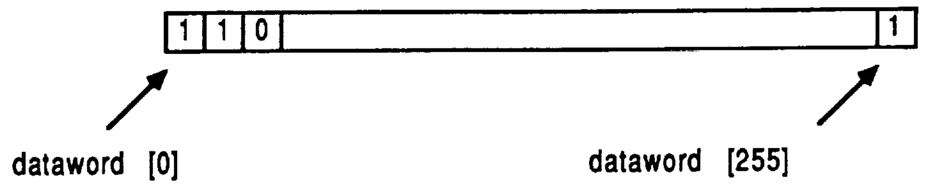
1. The Executive module passes the 256-bit reference address and the 256-byte "data" word to the Control module, as well as the cut-off Hamming distance to use.

2. The Control module passes the reference address and cut-off Hamming distance to the Address module, and the data word to the Stack module.

3. The Address module sequentially calculates the Hamming distance between each hard address and the reference address and compares it to the cut-off Hamming distance. Whenever it finds a distance less than or equal to the cut-off, it passes to the Control module both the tag of that hard address and the Hamming distance it calculated.

4. Whenever the Control module receives a tag from the Address module, it passes the tag to the Stack module and both the tag and Hamming distance to the Executive module. (The latter is to study performance characteristics of applications).

5. Whenever the Stack module receives a tag from the Control module, it performs 256 integer adds. Each add consists of adding sequential bytes from the data word (usually either $+1$ or -1) into sequential bytes in memory. A code segment to explain what happens is shown in Figure 1.18.



```
For each tag received from Control Module do :

begin

  StartingByteAddress := Tag * 256 ;
  for J := 0 to 255 do
      begin
      M := StartingByteAddress + J ;
     byte [M] := byte [M] + ( 2 * dataword [J] - 1 ) ;
    end  ;

end ;
```

The J^{th} component of the data word is accumulated into the J^{th} byte of each selected location; e.g. a '1' in the dataword causes the count to go up by 1, while a -1 causes the count to go down by 1. (The counters have a ceiling of $+127$ and a floor of -127).

Figure 1.18: What the Stack module does during a write operation.

```
Result :  Array [0..255] of Integer  (* an array of integers *)



For each tag received from Control Module do :

begin

  StartingByteAddress := Tag * 256 ;
  for J := 0 to 255 do
      begin
      M := StartingByteAddress + J ;
     Result [J] := Result [J] + byte [M] ;
    end  ;

end ;
```

The M^{th} counter of each selected location is accumulated into the M^{th} element of the array "Result" for J from 0 to 255.

Figure 1.19: How the Stack module accumulates the contents of locations selected by the Address module during a read operation.

SDM Read

1. The Executive module passes a reference address and a cut-off Hamming distance to the Control module.
2. The Control module passes both of these to the Address module.
3. The Address module performs exactly the same operations that it does during a write. Namely, it passes back to the Control module the tag and Hamming distance for every hard address Hamming distance from the reference address is less than the cut-off Hamming distance.
4. The Control module performs the same operations as during a write. Received tags and Hamming distances are passed to the Executive module, while the tags alone are also passed to the Stack module.
5. The Stack module establishes a 256 element integer array to hold the results of its operations. Every time it receives a tag from the Control module, it performs 256 integer adds; each add consists of adding a byte-sized counter value into an array element. The operation is shown in Figure 1.19.
6. When the Address module has gone through all 8192 hard addresses and the Stack module has performed its accumulation task for every selected tag, the Stack module sends to the Control module the results of its accumulations (i.e., Result[0] to Result[255] from Figure 1.19).
7. The Control module passes this result on to the Executive module.
8. The Executive module thresholds each result to either -1 or 1, based on a user-supplied threshold value. It then constructs a 256-component data word from the thresholded results. This data word is result of the read operation.

1.10 Summary

The prototype has been designed to provide a significant improvement in performance over software simulations of sparse distributed memory systems, while maintaining a high degree of flexibility. Those goals have been achieved by dividing the system into four modules, and by using standard subsystems (e.g., single-board computers) and software based implementations wherever feasible. One module contains a custom hardware based design. This was necessary in order to achieve the desired speed in the critical task of calculating and comparing Hamming distances.

The next chapter describes the hardware, and in particular the custom-designed Address module, in more detail.

Chapter 2

Hardware Description

This chapter describes the details of these hardware portions of the SDM system:

1. The Executive Module
2. The Control Module
3. The Stack Module
4. The Address Module

The Address module is described in the most detail since it uses custom hardware required to meet the performance specifications.

2.1 The Executive module

The primary requirements for the Executive module are:

- Provide a user interface to the SDM;
- Implement a high-speed communications protocol to the other portions of the SDM;
- Provide a common programmer interface;
- Handle the computation requirements for program modules that do not use the SDM.

The equipment that best fits the above requirements is the workstation. In particular, workstations provide an excellent user interface, have a programming interface with which most researchers are comfortable, and can handle the computation requirements.

Coincidentally, SDM simulators have been written for workstations; researchers who are familiar with these can easily adapt themselves to this implementation.

One component of critical importance is the communications interface between the Executive module and the SDM. The communications system must support the following operations:

1. Operation. The operational requirement is for 50 read/write operations per second. This target is based on about 25 hits per operation. Performance is very much a function of hit rate (see Flachs [2]). Each operation consists of the transfer of the 256-bit reference address, an operation identifier, and 256 32-bit sums per fold back to the Executive module.

2. Debugging. The entire contents of the SDM, including the Stack, Address, and Control module memories, should be transferable in a reasonable amount of time.

3. Setup. The contents of the Hard Address Memory, to be located on the Address module, should be quickly downloadable.

Of these, requirements 2 and 3 are for convenience; clearly, we want debug and set-up time to be a minimum. However, requirement 1 places a real bound on the communications subsystem.

Assuming the worst case for a single fold system, we must transfer 50 reference addresses, 50 instructions, and $50 * 256$ 32-bit sums per second. This translates into:

$$C = 50 * (256 + 8 + 256 * 32) = 422{,}800 \text{ bits/second}$$

Again assuming the worst case, we allow for a protocol overhead of 50%:

$$N = 2C = 634{,}200 \text{ bits/second}$$

for a single fold system. Additional folds add 409,600 bits/second/fold.[1]

One communications system that would easily meet this requirement is a bus-to-bus mapping switch. Clearly, the transfer rate of such a device greatly exceeds our requirement. However, the use of a bus map introduces an enormous amount of inflexibility into the SDM; in particular, preferential addresses will probably be already taken by the workstation. The use of a bus map would also require that the bus used by the Executive module be the same as that used by SDM. This limits flexibility of choice of the Executive module, and may place severe structural constraints on the SDM.

A solution to this problem is to use an existing communications protocol that is simple and provides the required bit rate. In particular, it helps to view the SDM as a smart disk drive that is attached to the Executive module. Most vendor-independent disk protocols provide read and write operations and support some level of control-message

[1] $2 * (50 * (256 * 16)) = 409,600$

handling. Moreover, most workstations support some form of vendor-independent disk protocol.

For our implementation, we chose the Small Computer Systems Interface protocol, or SCSI, to provide communications between the Executive module and the other modules of the SDM. SCSI supports a burst rate of 1.5 Mbps and provides a reasonable number of control functions. Because of the large number of workstations that support SCSI, this implementation of the SDM system may be attached to almost any workstation, mainframe, or even a personal computer.

Thus, the requirements for the Executive module indicate a powerful workstation that supports the SCSI protocol. Because of our familiarity with the products of Sun Microsystems, the current implementation uses a Sun 3/60 color workstation, with the SDM attached to one of the SCSI ports. We emphasize, however, that *any system that supports the SCSI protocol is capable of using the SDM as an attached processor.*

2.2 The Control module

The Control module, or CM, acts as the interface between the Executive module and the rest of the SDM. It manages the operation of the Address module and transfers tags from the AM to the appropriate fold in the Stack module.

The choice of a processor for the CM was heavily influenced by two considerations:

1. The need for a large (greater than 24 MB) address space;

2. The availability of software tools.

The large address space mandates the use of the VME-bus standard interconnect, which in turn favors the Motorola 68000 family of processors. In addition, software tools were readily available for this processor family. The memory-to-memory transfer bandwidth mandates the use of a 32-bit processor; the MC68030 fills all of these requirements.

Thus, the requirements for the CM are as follows:

1. Motorola 68030, 25 MHz or greater clock speed.

2. A large quantity (about 4MB) of dynamic RAM; to hold reference tables and Hard-Address-to-Tag translation maps.

3. Dual-ported memory to support asynchronous transfers.

A vendor search indicated that Force Computers would be able to deliver a board with these specifications, in addition to one that matched the requirements for the folds, which will be discussed later.

The actual functionality of the CM is implemented in software; however, there are a few features of the CM hardware that particularly facilitate the SDM:

1. Address module simulation. Because of the large quantity of memory available, a significant number of simulated AM operations can be performed on the CM.
2. Message Broadcast. The interrupt driven message broadcast system allows the Command module to issue commands to all stack modules simultaneously and await their response without polling.

2.3 The Stack module

The Stack module consists of a number of submodules, known as folds. Each fold is independent; thus, the design of the SM is simplified by considering it as made up of a number of similar submodules. The requirements for each fold are:

1. Enough memory to hold a 16-bit count (two bytes) for each bit in every word of the Hard Address Memory. Practically, this implies a need for $512 * 8,192 = 4$MB of memory as a minimum.
2. The memory must be dual ported for debugging purposes.
3. A processing system that will be able to perform integer additions quickly enough to satisfy the 50 operations/second requirement.

These requirements fall within the capabilities of a single-board computer, as long as the processor on the board is fast enough to perform the necessary integer adds.

To this end, considering the implementation decisions for the Control module, we again selected a processor card based on the Motorola MC68000 architecture. Force Computers was able to provide a board with a 25 MHz MC68030, and 4MB of dual ported dynamic RAM.

2.4 The Address module

The Address module, or AM, is the only custom component of the SDM. It will be described in considerably more detail than the other components. As mentioned in the architecture section, more than one AM may be used within the SDM if a performance increase is required. The reason for a custom implementation lies in the fundamental operation that the AM performs, which is a 256-bit Hamming distance calculation.

It should be stressed that a 256-bit word is *very* large. In order to fully appreciate the difficulty of designing a processor with 256-bit internal data paths, consider the following:

Table 2.1: Register addresses.

Register	Longname	[width]	R/W	#32-bit Regs	Reg#
TR	Tag Register	[16]	W	1	0
A 13-bit "tag" which locates a hard address in the Hard Address Memory.					
HAR	Hard Address Register	[256]	W	8	1–8
Enters a hard address into the Hard Address Memory.					
RAR	Reference Address Reg.	[256]	W	8	9–16
Sets the reference address—the Address in question.					
MR	Mask Register	[256]	W	8	17–24
Sets the Hamming AND mask.					
LR	Limit Register	[8]	W	1	25
Defines the radius of the Hamming Sphere. Any tag with Hamming distance less than or equal to this will be written to the Tag Cache.					
CSR	Command/Status Register	[8]	R/W[a]	1	26
The bits in this register determine the current state of the machine.					
TCO	Tag Cache[b] Output	[24]	R	1	27

[a] The Command/Status Register has different interpretations when being read or written.
[b] A Tag Cache is a 256-word cache that holds the accepted tags.

1. If one designs the data-path elements with byte-wide, commercially available products, each element in the data path requires 32 chips.

2. Each link in the data path requires 256 connections; a four-element data path will easily require in excess of 1,000 wire wraps.

These two numbers (components and connections) will quickly outstrip the capacity of most boards and development systems; our system will fit only on a 400 mm high by 366 mm wide 9U sized VME bus card, which is the largest one available.

2.4.1 Overview and board floorplan of Address module

The Address module appears to the Control module as a set of sequential memory locations on the VME bus. These addresses are defined in Table 2.1.

Reading the Command/Status Register returns the following bits:

Bits	6	5	4	3	2	1	0
	FULL	DONE	MODE	EMPTY	RST	ST1	ST0

These bits are defined as follows:

STATE: Meaning:

ST1	ST0	
0	0	Reset
0	1	Running
1	0	Hit
1	1	Wait/Done

Reset: (=1 for 200ms at power-on)

RST	
0	Ready
1	Reset

Tag Cache Empty Indicator:

EMPTY	
0	TCO not empty
1	TCO empty

Complemented Mode:

MODE	
0	Uncomplemented Addresses
1	Complemented Addresses

Done Indicator:

DONE	
0	AM not done processing
1	AM done processing

Full Indicator:

FULL	
0	Tag cache is full
1	Tag cache is not full

Writing the Command/Status Register affects the following bits:

Bits	7	6	5	4	3	2	1	0
	X	X	X	CMODE	FORCE	FRST	FST1	FST0

These bits are defined as follows:

X: DON'T CARE

FORCED STATE: When the FORCE bit is set, the AM is forced into a particular state until the FORCE bit is reset. This is used for debugging purposes only.

FORCE	FST1	FST0	Operation:
0	X	X	Normal operation
1	0	0	Reset
1	0	1	Running
1	1	0	Hit
1	1	1	Wait/Done

Forced Reset: Setting this bit forces the AM to be reset.

FRST	
0	Ready
1	Reset

Complemented Mode:

CMODE	
0	Use uncomplemented addresses
1	Use complemented addresses

Bits in the tag cache output are defined as follows:

Bit23 – Bit16	Hamming Distance
Bit15 – Bit0	Tag ID

Figure 2.1 is the board floorplan for the address module. It also names the six submodules of the AM, namely:

1. Clock and Sequencer: This submodule contains the master state machine and the master clock.

2. Hard Address Memory. All 8,192 256-bit addresses are contained in this memory, which consists of 32 8KB static RAMs.

3. Arithmetic Logic Unit: This submodule performs the 256-bit exclusive-OR and mask operations. The Reference Address Register and Mask Register, each consisting of 32 8-bit latches, are combined here with a hard address by a logic unit, which consists of 64 programmable logic arrays (PLAs). The ALU result is a 256-bit quantity with a 1 in each bit position where the reference address differs from the hard address and is not masked by a 0 in the corresponding bit position in the Mask Register. The *number* of 1s in this result is the Hamming distance between the Reference and hard address. (The Mask is a 256-bit user-specified pattern to restrict the Hamming distance calculation to a subset of the 256 bits, if desired.) The CMODE bit in the Command/Status Register causes the ALU to use the complemented hard address.

4. The Bit Counter. The bit counter calculates the Hamming distance by adding the 256 bits of the ALU output and then compares the result with the Limit Register.

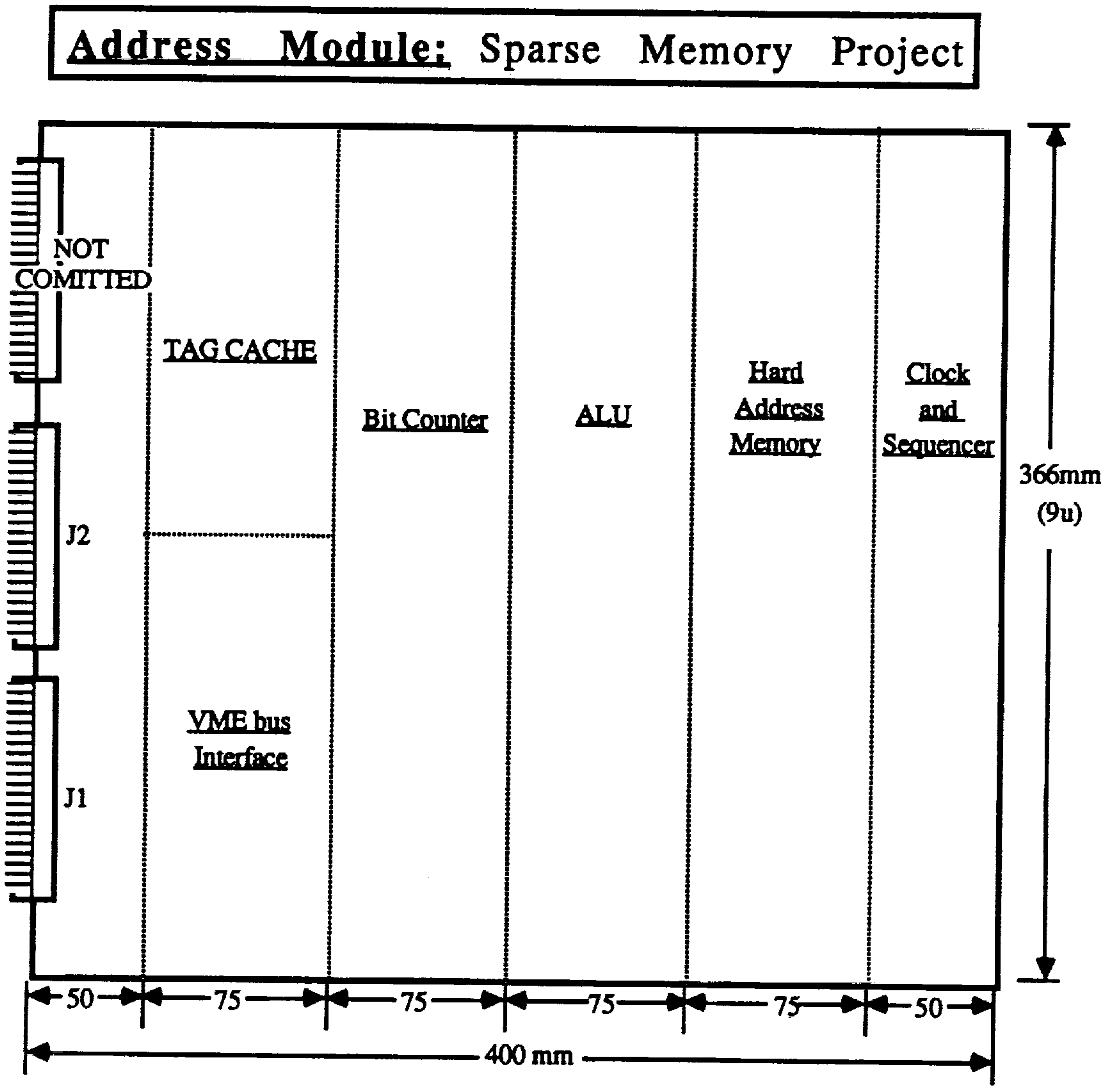


Figure 2.1: Floorplan.

If the result is less than or equal to the Limit Register, there is a "hit," and both the tag associated with the particular hard address and its Hamming distance from the reference address are written into the Tag Cache.

5. Tag Cache. After a matching address is detected, its target Hamming distance must be fetched by the control module. The Tag Cache acts as a first in–first out buffer between the address module and the control module, allowing the two modules to operate asynchronously. The buffer is 256 tags long, so if the command module can keep up, the matching process can continue to completion with very little idle time. If, on the other hand, the address module outruns the control module and the cache fills up, the address module enters the wait state until the cache is emptied. Once the cache is cleared the address module re-enters the RUN state and continues operation.

6. Bus Interface. Finally, the bus interface provides address mapping, power, and control information to the AM from the VME bus.

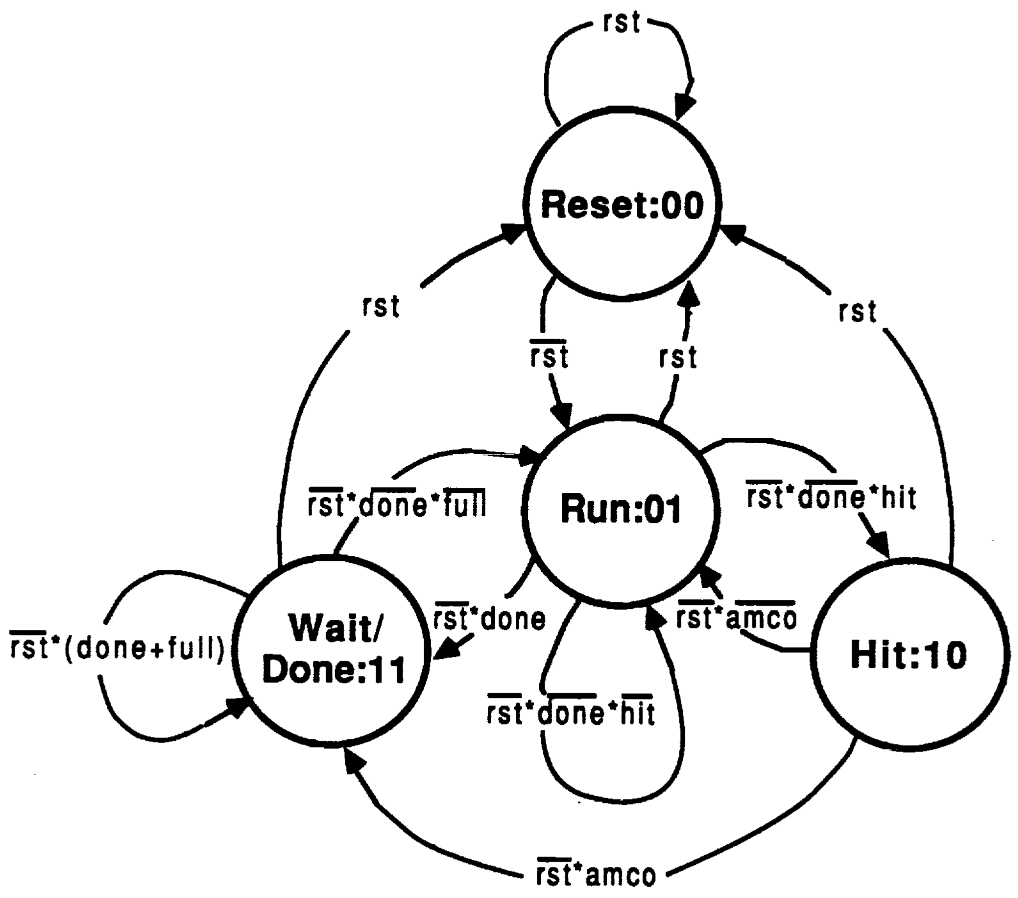
Figure 2.2 details the state transitions of the Address module. The four processor states are as follows:

0. **RESET.** This is the power-up state of the AM. However, the Address module will enter the RUN state and begin processing garbage data once the initial power-on reset mechanism has finished. Before writing the Hard Address Memory or in any other way initializing the AM, the Control module must put the AM in the RESET state by setting the FRST bit in the Command/Status Register. The AM will then remain in the reset state until the FRST bit is reset, causing the AM to enter the RUN state and begin processing the reference address.

1. **RUN.** The AM enters this state from a reset when the FRST bit in the Command/Status Register is zeroed. It continues to loop within this state until a hit (hard address "close" to Reference) occurs or until the Hard Address Memory is exhausted.

2. **HIT.** When a hit occurs, the AM enters the HIT state and writes the tag associated with the hard address and its Hamming distance into the Tag Cache. This state is included for timing reasons, enabling the Address module to process data at a high frequency (1 MHz clock rate).

3. **WAIT/DONE.** This state is entered when the Hard Address Memory has been fully examined or when the Tag Cache is full (has data in its last location that has not been read by the Control module). The DONE status bit allows the Control module to differentiate between these two conditions. The AM will leave the WAIT state and resume the RUN state when the CM reads the last location in the Tag Cache. The AM will remain in the DONE state until the CM sets the


rst
Reset:00
rst
rst
$\overline{rst}$
rst
Run:01
$\overline{rst}$*$\overline{done}$*$\overline{full}$
$\overline{rst}$*$\overline{done}$*hit
$\overline{rst}$*(done+full)
Wait/ Done:11
$\overline{rst}$*done
$\overline{rst}$*$\overline{amco}$
Hit:10
$\overline{rst}$*$\overline{done}$*$\overline{hit}$
$\overline{rst}$*amco

Signal Description:

rst	The value of the state register reset line.
hit	Is high when the Hamming distance is less than or equal to the limit.
amco	Tag Cache will be full at the next positive clock edge.
full	Tag Cache is full.
done	The entire address space has been searched.

Figure 2.2: State transition diagram of Address module.

FRST status bit, forcing the AM into the RESET state during which it can be set up for more reference address processing.

We will now examine each submodule of the AM in detail.

2.4.2 The Clock/Sequencer

The Clock/Sequencer implements the state machine described above. It consists of a registered PAL to implement the state transitions and a 1.0 MHz clock. It also contains the Reset flip-flop.

The Command/Status Register allows the state of the AM to be determined for debugging purposes. In effect, the state machine delivers a hardwired "instruction" to the rest of the AM; note that the four states correspond to programming statements, while the state-transition logic defines the transfer of control in the microprogram.

The clock/sequencer submodule also contains a 13-bit cyclical counter which is used to address the Hard Address Memory during a Run cycle.

2.4.3 The Hard Address Memory

The Hard Address Memory stores all 8,192 256-bit hard addresses, which define the memory distribution. It consists of 32 8K-by-8 bit static CMOS RAMs along with the associated internal bus-switching logic that demultiplexes the input/output of the RAMs.

Associated with each 256-bit hard address is the 13-bit tag that locates it in the Hard Address Memory. In order for the Control module to load a hard address, it must first load the appropriate tag into the Tag Register (implemented with two 8-bit latches). The CM then writes the hard address into the 8 32-bit registers that comprise the Hard Address Register. From there it is loaded into the CMOS static RAMs that comprise the Hard Address Memory, thus storing a complete 256-bit quantity in the Hard Address Memory.

2.4.4 The ALU

The ALU consists of a Reference Address Register, a Mask Register, and a logic unit to perform a 256-bit XOR-AND operation. The registers consist of 32 8-bit latches each, and the logic unit is implemented in 64 programmable array logic chips (PALs). In addition, the ALU submodule houses the "Complement Mode" flip-flop bit. This allows the complementary address space to be searched without reloading the Hard Address Memory.

2.4.5 The Bit Counter

The resulting 256-bit word from the ALU is then passed to the Bit Counter. The number of bits in this word that are '1' is the Hamming distance between the reference address and the hard address in question.

The Bit Counter implements a 256-bit-wide one-bit adder in four stages. Each stage was constructed from several 32KB EPROMs. The first stage consists of 17 such EPROMs, with each of the 15 EPROM address lines connected to one of the 256 bits from the ALU output. The resulting four-bit outputs are then added in the next stage, and so forth. At the end of the fourth stage, the bit count is represented by a single 8-bit number. Note that if the Hamming distance is 256 it will be encoded as 255 since eight bits can only represent 0 through 255. This necessary anomaly was determined to have no practical effect on the usefulness of the Sparse Distributed Memory System.

This eight-bit result is then compared with the Limit Register (a single 8-bit latch). If the result is less than or equal to the limit, the Hit status bit is set and the AM will enter the HIT state, storing the tag and the Hamming distance into the Tag Cache.

2.4.6 The Tag Cache

The central components of the Tag Cache are a trio of dual-ported 256 byte static RAMs. One RAM stores the Hamming distance, the other two store the tag. In operation, an internal write counter points to the next available address in the DPRAM where the tag and Hamming distance can be stored, while an internal read counter points to the next location in the Tag Cache to be read by the CM, thus effecting a FIFO. When the AM enters the HIT state, the tag and distance are written to the DPRAM, after which the internal write counter is incremented. The EMPTY status bit is then reset, signalling the Control module that the Tag Cache has data to be read. When the CM reads the Tag Cache, the internal read counter is incremented. When the read and write counters are equal, the Tag Cache is empty and the EMPTY status bit is set. It is important that the CM read the Tag Cache *only* when the EMPTY bit is zero, otherwise the operation of the AM will be unpredictable until it is once again reset by the CM or through powering off.

When the write counter wraps around again to 0, the AM enters the WAIT state until the CM has read the last location of the Tag Cache and the EMPTY status bit is once again set. The AM then enters the RUN state and continues execution until the entire Hard Address Memory has been examined or the Tag Cache has once again filled up.

2.4.7 The Bus Interface

Finally, the Bus Interface submodule handles all of the VME-bus addressing for the AM and internal to external bus translations. It consists of a set of bus transceivers along with a demultiplexor to generate all 28 register address lines, and supplies the necessary power to the board. Only Long Words (32-bit quantities) should be read or written to the AM. Any other data type may result in incorrect data transfers.

2.4.8 Operation of the Address module

The AM operates in two distinct modes. Before an application is run on the SDM system, specific hard addresses must be loaded into the AM and the various registers must be initialized. This is accomplished in the RESET state as shown in the state transition diagram, Figure 2.2.

Once set-up is complete, applications can be run on the system. Application processing begins when the Control module takes the Address module out of RESET mode and into RUN mode by clearing the RST status bit in the Command/Status Register. A single read or write for an application causes the AM to:

1. Execute the Run-state 8192 times. During each execution, the reference address is compared to one of the hard addresses, and the Hamming distance between them is calculated and compared to the value in the Limit Register.
2. Visit the HIT state once for every hit encountered during the RUN state.
3. Visit the WAIT state whenever the tag cache gets filled.
4. Visit the DONE state when the entire contents of the Hard Address Memory have been examined.

Note that the WAIT state and DONE state are identical except for the value of the DONE status bit.

The procedure for running an application is as follows.

Set-up:

1. The CM sets the RST bit in the Command/Status Register, placing the AM in the RESET state.
2. The CM loads the Hard Address Memory. This is accomplished by first loading the Tag Register with the appropriate tag, and then loading the Hard Address Register. The 256-bit Hard Address Register consists of eight 32-bit registers which can be loaded in any order. This procedure is repeated once for each hard address.
3. The CM loads the other registers: the Mask Register, the Limit Register and the Reference Address Register. The order of access is not important. Also, values loaded into the registers and the Hard Address Memory will remain intact even after the AM has been placed in the RESET state. The registers and the Hard Address Memory will contain random data, however, after a power up.

Operating:

1. The CM clears the RST bit in the Command/Status Register, and the state machine enters the RUN state.
2. During each RUN cycle, the next hard address in the Hard Address Memory (beginning at tag 0 and incrementing to tag 8191) is transferred to the ALU, to be XORed with the reference address. The result is then ANDed with the Mask Register and passed to the Bit Counter. The result propagates through all four stages of the Bit Counter and is then compared to the Limit Register. If the result is less than or equal to the Limit, the HIT status bit is set and the AM enters the HIT state. If there is no hit, the AM continues in the RUN state, examining the next hard address. When the last hard address has been examined, the AM enters the DONE state.
3. During a HIT cycle, the Hamming distance is sent to the Tag Cache, along with the hard address tag. If the Tag Cache is full, the AM enters the WAIT state. If the last hard address has been examined, the AM enters the DONE state. Otherwise the AM resumes execution in the RUN state.
4. During a WAIT cycle, the AM suspends operation and waits until the CM has read the last entry in the Tag Cache. At that point, if all hard addresses have been examined, the AM enters the DONE state. Otherwise the AM resumes execution in the RUN state.
5. During a DONE cycle, the AM asserts the DONE status bit and performs no operation until the CM sends it into the RESET state by setting the RST status bit, or by a power-on reset.

2.4.9 Additional hardware

Beyond the basic board set and workstation, a VME bus card cage and SCSI cable are also needed. The card cage accommodates at least two 400mm 9U sized cards and more than eleven 6U 200mm cards. The cable required depends on the workstation connectors.

Chapter 3

Use of the SDM

3.1 Programmer's Interface

The Programmer's Interface provides access to the SDM from Sun applications. Although programs can be written that directly access the raw SCSI device, it is suggested that the Programmer's Interface be used for greater modularity and abstraction. If the details of the SCSI transactions between the Executive Module and the Control Module are modified, it should be necessary only to update the Programmer's Interface and then recompile the applications.

3.1.1 Data Types

In the descriptions that follow we shall make use of the data types *foldentry, dataword, addressvec, addressbits, tagaddress,* and *foldset.* These types are defined in the file `sdmcomm.h` and are subject to change with future modification of the system. The current definitions are as follows:

- foldentry is an array of 256 short (16 bit) integers.
- dataword is an array of 256 (32 bit) integers.
- addressvec is an array of 32 unsigned characters used to store a 256 bit address as a bit vector.
- addressbits is an array of 256 characters used to store a 256 bit address as a Boolean array.
- tagaddress is an unsigned short, only 13 bits of which are currently used (the least significant bits), that represents a tag. The valid range of tags is 0 to NUM_TAGS-1, where NUM_TAGS is currently defined in `sdmcomm.h` to be 8192.

- foldset is an unsigned integer used as a bit vector. Each bit corresponds to one fold so the number may be used to represent a single fold or a collection of folds. The following constants are defined in **sdmcomm.h** to reference folds:

NUM_FOLDS	4
FOLD(0)	1
FOLD(1)	2
FOLD(2)	4
FOLD(3)	8
ALL_FOLDS	15

3.1.2 Overview of Routines

The Programmer's Interface consists of a set of C procedures, **sdmcomm.c**, and a header file of external declarations, **sdmcomm.h**. The following procedures are currently provided:

- OpenSparse, ResetSparse, SelectFold and CloseSparse are used to acquire and release the SCSI port and to modify the general state of the system.
- SparseAddress, ShiftAddress, SparseRead, SparseReadVec, SparseWrite, and SparseWriteVec are the primary Sparse operations.
- SetLimit and ReadLimit provide access to the current limit on the Hamming radius.
- SetThreshold and ReadThreshold provide access to the current value of the threshold used in the conversion of solution from a dataword (consisting of 32 bit integers) to a bit vector.
- SetMask and ReadMask provide access to the current address mask.
- SetCompMode and ReadCompMode provide access to the activation of the complemented address mode.
- SetFoldEntry, ReadFoldEntry, FillFold, ZeroEntireFold, SetHard, ReadHard, FillHard, and ZeroAllHard provide direct access to the fold data and the hard address registers.
- SaveSparse and RestoreSparse cause portions of the state of the Sparse to be saved to or restored from a data file on the Sun.
- packaddr and unpackaddr are utilities for the conversion between the Boolean array and bit vector representations of addresses.
- SetDebug is used for debugging.

3.1.3 Descriptions of Routines

- int OpenSparse()

  int CloseSparse()

  OpenSparse performs an open on the SCSI device for read/write access. A single, successful OpenSparse must be executed before any other Sparse operation is attempted. CloseSparse, the complement to OpenSparse, performs a close on the SCSI device, making it available to other applications.

  Both OpenSparse and CloseSparse return a value of 0 upon successful completion. If an error occurs, the value -1 is returned and external variable **errno** is set to indicate the cause of the error.

- void ResetSparse()

  ResetSparse performs the following actions:

  - places the Address Module in "Forced Reset" mode (see *Address Module Hardware Guide*),
  - clears the complemented address mode flag (see `SetCompMode()`),
  - selects all of the folds (see `SelectFold()`),
  - sets the tag bounds to their maximum extent, 0 and NUM_TAGS-1, (see `SetTagBounds()`), and
  - clears the tags sets for all of the folds.

  ResetSparse should normally be called immediately after OpenSparse.

- void SelectFold(f)
  foldset f;

  void ReadSelectFold(f)
  foldset *f;

  SelectFold specifies which folds will be active during subsequent operations such as SparseAddress, SparseWrite, SparseRead, and SetFoldEntry. The foldset `f` (see the description of `foldset` above) may specify any number of folds from zero to four. For example:

  `SelectFold(FOLD(0))` sets fold 0 to be the only active fold,

  `SelectFold(FOLD(0)|FOLD(1))` sets folds 0 and 1 to active, and

  `SelectFold(ALL_FOLDS)` activates all four folds.

  ReadSelectFold sets `f` to the currently active foldset.

- void SparseAddress(reference_address)
  addressvec reference_address;

Sets the tag set of each the currently selected folds to consist of those tags whose corresponding hard addresses match **reference_address**[1]. See SelectFold, SetMask, SetLimit, SetTagBounds, and SetCompMode.

- void ShiftAddress(reference_address)
  addressvec reference_address;

  Sets the tag set of fold 0 to consist of those tags whose corresponding hard addresses match **reference_address**[1], and shifts the previous tag set of fold 0 to fold 1, of fold 1 to fold 2, and of fold 2 to fold 3. Note that SelectFold does not affect upon the operation of the ShiftAddress routine.

- int SparseWrite(data)
  dataword data;

  int SparseWriteVec(data)
  addressvec data;

  Executes a Sparse Write operation to each of the currently selected folds using their current tag sets and specified data word. Returns the total number of hits[2].

  SparseWriteVec is exactly the same as SparseWrite except that **data** is a bit vector indicating increment or decrement in the corresponding counters.

- int SparseRead(result)
  dataword result;

  int SparseReadVec(result)
  addressvec result;

  SparseRead executes a Sparse Read operation from the currently selected folds and stores the resulting dataword in **result**. Returns the total number of hits[2]. If the number of hits is zero, then **result** will contain all zeroes.

  SparseReadVec is exactly the same as SparseRead except that **result** is a bit vector indicating increment or decrement in the corresponding counters.

- void SetLimit(limit)
  int limit;

  void ReadLimit(limit)
  int *limit;

  SetLimit sets the solution threshold on the Hamming distance to **limit**, and ReadLimit sets **limit** to the current solution threshold.

[1]Matches computed with a proper regard for the address mask, the solution threshold, the tag bounds, and the complemented address mode flag.

[2]The total number of hits can be regarded as the total number of fold entries contributing to the result or, equivalently, as the sum over the selected folds of the sizes of their tag sets.

- void SetThreshold(th)
  int th;

  void ReadThreshold(th)
  int *th;

  SetThreshold sets the threshold used by `SparseReadVec` to the value `th`. A bit in the vector returned by `SparseReadVec` will contain a 1 if and only if the corresponding position in resulting dataword is *greater* than this threshold. ReadThreshold sets `th` to the current threshold.

- void SetMask(mask)
  addressvec mask;

  void ReadMask(mask)
  addressvec mask;

  SetMask sets the address mask used in the computation of matching addresses to `mask`. A zero in the mask indicates that the corresponding position is to be ignored during the determination of a match. ReadMask sets `mask` to the current value of the address mask.

- void SetCompMode(bool)
  int bool;

  int ReadCompMode()

  SetCompMode sets the value of the complemented address mode flag. If the value of `bool` is zero then the flag is cleared, otherwise the flag is set so that complemented address mode is used (see *Address Module Hardware Guide*). ResetSparse() also clears this flag.

  ReadCompMode determines whether the complemented address mode flag is set (return 1 if set, 0 otherwise.)

- void SetTagBounds(low,high)
  tagaddress low, high;

  ReadTagBounds(low,high)
  tagaddress *low, *high;

  SetTagBounds sets the low tag bound to `low` and the high tag bound to `high`. During the SparseAddress and ShiftAddress operations, a hard address can only be considered a hit if its tag is greater than or equal to the low tag bound and if its tag is less than or equal to the high tag bound. In this way, address matching can be restricted to a subset of the hard addresses in the Address Module.

  ReadTagBounds sets `low` and `high` to the current tag bounds.

  `SetTagBounds(0,NUM_TAGS-1)` removes this restriction as to which hard addresses can be hit.

- void SetFoldEntry(tag,entry)
  tagaddress tag;
  foldentry entry;

  void FillFold(from,to,pattern)
  tagaddress from, to;
  unsigned char pattern;

  void ZeroEntireFold()

  void ReadFoldEntry(tag,entry)
  tagaddress tag;
  foldentry entry;

  SetFoldEntry writes `entry` into the fold entry corresponding to hardware tag `tag` in those folds that are currently active. This operation would typically be used only to initialize the fold.

  FillFold takes the byte `pattern` and replicates it in the fold memory of the currently active folds to fill all of those fold entries with tags from `from` up to and including `to`. Like SetFoldEntry, this operation would typically be used only to initialize the folds.

  ZeroEntireFold sets *all* fold entries to zeroes.

  ReadFoldEntry retrieves a fold entry corresponding to hardware tag `tag` in the fold that is currently active and stores the result in `entry`. The result is unspecified if more than one fold is currently active.

- void SetHard(tag,value)
  tagaddress tag;
  addressvec value;

  void FillHard(from,to,pattern)
  tagaddress from, to;
  unsigned char pattern;

  void ZeroAllHard()

  void ReadHard(tag,value)
  tagaddress tag;
  addressvec value;

  SetHard sets the hard address `value` to correspond to the tag value `tag`.

  FillHard takes the byte `pattern`, replicates to it to form a hard address, and then does a SetHard with this address and tags `from` through `to`, inclusive.

  ZeroAllHardsets *all* hard addresses to all zeroes. This has the same effect as FillHard(0,NUM_TAGS-1,0x00) but is very fast.

  ReadHard sets `value` to the hard address previously stored with `tag`. Because the hard address registers of the address module are write-only, the value is obtained from a shadow register which is updated on every SetHard, ZeroAllHard and RestoreSparse. The value in the shadow register is not guaranteed to be identical to that used by the address module. In particular, the actual values are unknown at power up. ZeroAllHard() can be used to put the registers into a known state.

- int SaveSparse(filename)
  char *filename;

  int RestoreSparse(filename)
  char *filename;

  SaveSparse creates a file with the specified filename and stores the majority of the current state of the Sparse (the limit, mask, threshold, hard addresses, and fold entries) into the file. RestoreSparse can then be used to restore that state from the file.

  NOTE: The file will be large (over 16MB with four folds) and the operation can take a long time.

- packaddr(bool,vec)
  addressbits bool;
  addressvec vec;

  unpackaddr(vec,bool)
  addressvec vec;
  addressbits bool;

  Although none of the Sparse transactions use the Boolean array representation of the 256 bit addresses, it may be easier for programmers to manipulate addresses in this format. The unpackaddr routine converts an addressvec (bit vector) into an addressbits array where each element is a one if the corresponding bit was set, zero otherwise. The packaddr routine does the inverse conversion, setting only those bits whose corresponding elements are not zero.

- void SetDebug(level)
  int level;

  Sets the debugging level on the Sparse to `level`. The debugging level specifies the verbosity of the control module. The default level is NULL, which generates a minimum of messages.

Bibliography

[1] P. A. Chou. Capacity of the Kanerva associative memory. *IEEE Transactions on Information Theory*, 35:281–298, March 1989.

[2] B. K. Flachs. Evaluation of SDM prototype. Technical report, Computer Systems Laboratory, Stanford University, 1990. In preparation.

[3] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, USA*, 79:2554–2558, April 1982.

[4] P. Kanerva. *Sparse Distributed Memory.* MIT Press, Cambridge, MA, 1988. A Bradford Book.

[5] J. D. Keeler. Comparison between Kanerva's SDM and Hopfield-type neural network models. *Cognitive Science*, 12:299–329, December 1988.